USENIX Association

Proceedings of the
2019 USENIX Annual Technical Conference

July 10–12, 2019
Renton, WA, USA

© 2019 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 978-1-939133-03-8

Conference Organizers

Program Co-Chairs

Dahlia Malkhi, *VMware Research*

Dan Tsafrir, *Technion—Israel Institute of Technology & VMware Research*

Program Committee

Nitin Agrawal, *ThoughtSpot*

Irfan Ahmad, *Magnition*

Deniz Altinbuken, *Google*

Nadav Amit, *VMware Research Group*

Saurabh Bagchi, *Purdue University*

Mahesh Balakrishnan, *Yale University and Facebook*

Antonio Barbalace, *Stevens Institute of Technology*

Andrew Baumann, *Microsoft Research Redmond*

Adam Belay, *Massachusetts Institute of Technology*

Ken Birman, *Cornell University*

Edward Bortnikov, *Yahoo*

Herbert Bos, *Vrije Universiteit Amsterdam*

Andre Brinkmann, *Johannes Gutenberg-University Mainz*

Edouard Bugnion, *École Polytechnique Fédérale de Lausanne (EPFL)*

Randal Burns, *Johns Hopkins University*

Anton Burtsev, *University of California, Irvine*

Haibo Chen, *Shanghai Jiao Tong University*

Vijay Chidambaram, *The University of Texas at Austin and VMware Research*

Asaf Cidon, *Barracuda Networks*

Austin Clements, *Google*

David Cock, *ETH Zurich*

Paolo Costa, *Microsoft Research*

John Criswell, *University of Rochester*

Charlie Curtsinger, *Grinnell College*

Dilma Da Silva, *Texas A&M University*

Nathan Dautenhahn, *Rice University*

Eyal de Lara, *University of Toronto*

Christina Delimitrou, *Cornell University*

Angela Demke Brown, *University of Toronto*

Fred Douglis, *Perspecta Labs*

Eric Eide, *University of Utah*

Michael Factor, *IBM Research—Haifa*

Pascal Felber, *University of Neuchâtel*

Christof Fetzer, *TU Dresden*

Moshe Gabel, *University of Toronto*

Ada Gavrilovska, *Georgia Institute of Technology*

Phillip Gibbons, *Carnegie Mellon University*

Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

Ashvin Goel, *University of Toronto*

Boris Grot, *University of Edinburgh*

Steven Hand, *Google*

Liting Hu, *Florida International University*

Yu Hua, *Huazhong University of Science and Technology*

Jian Huang, *University of Illinois at Urbana-Champaign*

Bill Jannen, *Williams College*

Sudarsun Kannan, *Rutgers University*

Manos Kapritsos, *University of Michigan*

Kimberly Keeton, *Hewlett Packard Labs*

Samira Khan, *University of Virginia*

Taesoo Kim, *Georgia Institute of Technology*

Sam King, *University of California, Davis*

Aasheesh Kolli, *The Pennsylvania State University*

Dejan Kostic, *KTH Royal Institute of Technology*

Geoff Kuenning, *Harvey Mudd College*

Patrick P.C. Lee, *The Chinese University of Hong Kong*

Xing Lin, *NetApp*

Ethan Miller, *University of California, Santa Cruz, and Pure Storage*

Changwoo Min, *Virginia Polytechnic Institute and State University*

Adam Morrison, *Tel Aviv University*

Gilles Muller, *Inria*

Dushyanth Narayanan, *Microsoft Research*

David Nellans, *NVIDIA*

Ed Nightingale, *Microsoft Research*

Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*

Aurojit Panda, *New York University*

Peter Pietzuch, *Imperial College London*

Don Porter, *The University of North Carolina at Chapel Hill*

Michael Reiter, *The University of North Carolina at Chapel Hill*

Scott Rixner, *Rice University*

Timothy Roscoe, *ETH Zurich*

Chris Rossbach, *The University of Texas at Austin and VMware Research*

Leonid Ryzhyk, *VMware Research*

Bianca Schroeder, *University of Toronto*

Liuba Shrira, *Brandeis University*

Keith A. Smith, *NetApp*

Patrick Stuedi, *IBM Research*

Michael Stumm, *University of Toronto*

Ryan Stutsman, *University of Utah*

Steve Swanson, *University of California, San Diego*

Michael Swift, *University of Wisconsin–Madison*

Nisha Talagala, *Pyxeda AI*

Theodore Ts'o, *Google*

Chia-Che Tsai, *Texas A&M University*
Joseph Tucek, *Amazon*
Haris Volos, *Google*
Marko Vukolic, *IBM Research Zurich*
Carl Waldspurger, *Carl Waldspurger Consulting*
Ric Wheeler, *Facebook*
Dan Williams, *IBM T.J. Watson Research Center*
Youjip Won, *Korea Advance Institute of Science and Technology (KAIST)*
Gala Yadgar, *Technion—Israel Institute of Technology*
Yuval Yarom, *University of Adelaide and Data61*
Erez Zadok, *Stony Brook University*
Zheng Zhang, *Rutgers University*

Extended Review Committee

Irina Calciu, *VMWare Research Group*
Orr Dunkelman, *University of Haifa*
Ittay Eyal, *Technion—Israel Institute of Technology*
David Grove, *IBM Research*
Ajay Gulati, *ZeroStack*
Tim Harris, *Amazon UK*
Gernot Heiser, *University of New South Wales*
Asim Kadav, *NEC Labs*
Julia Lawall, *Inria/LIP6*
Kfir Lev-Ari, *Apple*
Carlos Maltzahn, *University of California, Santa Cruz*
Jason Nieh, *Columbia University*
Erik Riedel, *EMC*
Mark Silberstein, *Technion—Israel Institute of Technology*
Animesh Trivedi, *Vrije Universiteit Amsterdam*
Ymir Vigfusson, *Emerson University*
Lluis Vilanova, *Technion—Israel Institute of Technology*
Yang Wang, *Ohio State University*
Michael Wei, *VMWare Research Group*
Keith Winstein, *Stanford University*
Guoqing (Harry) Xu, *University of California, Los Angeles*
Noa Zilberman, *University of Cambridge*

Best of the Rest Session Co-Chairs

Amy Tai, *VMware Research*
Chia-Che Tsai, *Texas A&M University*

Lightning Talks Co-Chairs

Deniz Altinbuken, *Google*
Aasheesh Kolli, *The Pennsylvania State University and VMware Research*

Submissions Co-Chairs

Lalith Suresh, *VMware Research*
Gerd Zellweger, *VMware Research*

External Reviewers

Divyakant Agrawal
Mohammad Alizadeh
Jia-Ju Bai
Hitesh Ballani
Tom Barbette
Kirill Bogdanov
Mihai Budiu
Aaron Carroll
Mosharaf Chowdhury
Israel Cidon
Tudor David
Quentin De Coninck
Giovanni Di Crescenzo
Aleksandar Dragojevic
Haggai Eran
Anshul Gandhi
Manya Ghobadi
Amir Gholaminejad
Adrien Ghosn
James Gleeson
William Hatch
Shachar Itzhaky
Anand Iyer
David M. Johnson
Rob Johnson
Antonis Katsarakis
Eric Keller
Marios Kogias
Richard Li
Shan Lu
Peter Macko
Haohui Mai
Ketan Mayer-Patel
Arif Merchant
Ben Pfaff
Mia Primorac
Amna Shahab
Philip Shilane
Dongkun Shin
Alex Shraer
Igor Smolyar
Weijia Song
Robert Soule
Angelos Stavrou
Swami Sundararaman
Amy Tai
Dmitrii Ustiugov
Shivaram Venkataraman
Udi Wieder
Matei Zaharia
Bowen Zhou

USENIX ATC '19: 2019 USENIX Annual Technical Conference

July 10–12, 2019 Renton, WA, USA

Refreshing ATC – USENIX ATC '19 Program Co-Chairs Message xi

Real-World, Deployed Systems

The Design and Operation of CloudLab 1
Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, and Kirk Webb, *University of Utah;* Aditya Akella, *University of Wisconsin–Madison;* Kuangching Wang, *Clemson University;* Glenn Ricart, *US Ignite;* Larry Landweber, *University of Wisconsin–Madison;* Chip Elliott, *Raytheon;* Michael Zink and Emmanuel Cecchet, *University of Massachusetts Amherst;* Snigdhaswin Kar and Prabodh Mishra, *Clemson University*

Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure 15
Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler, *Oracle Corporation*

Zanzibar: Google's Consistent, Global Authorization System 33
Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, and Nina Kang, *Google;* Lea Kissner, *Humu, Inc.;* Jeffrey L. Korn, *Google;* Abhishek Parmar, *Carbon, Inc.;* Christina D. Richards and Mengzhi Wang, *Google*

IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services 47
Biswaranjan Panda and Deepthi Srinivasan, *Nutanix Inc.;* Huan Ke, *University of Chicago;* Karan Gupta and Vinayak Khot, *Nutanix Inc.;* Haryadi S. Gunawi, *University of Chicago*

Runtimes

PARTISAN: Scaling the Distributed Actor Runtime 63
Christopher S. Meiklejohn and Heather Miller, *Carnegie Mellon University;* Peter Alvaro, *UC Santa Cruz*

Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation 77
Changheng Song, *Fudan University;* Wenwen Wang, Pen-Chung Yew, and Antonia Zhai, *University of Minnesota;* Weihua Zhang, *Fudan University*

Transactuations: Where Transactions Meet the Physical World 91
Aritra Sengupta, Tanakorn Leesatapornwongsa, and Masoud Saeida Ardekani, *Samsung Research;* Cesar A. Stuardo, *University of Chicago*

Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code 107
Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha, *University of Massachusetts Amherst*

Filesystems

Extension Framework for File Systems in User space 121
Ashish Bijlani and Umakishore Ramachandran, *Georgia Institute of Technology*

FlexGroup Volumes: A Distributed WAFL File System 135
Ram Kesavan, *Google;* Jason Hennessey, Richard Jernigan, Peter Macko, Keith A. Smith, Daniel Tennant, and Bharadwaj V. R., *NetApp*

EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices 149
Xiang Gao, *Huawei Technologies Co., Ltd.;* Mingkai Dong, *Shanghai Jiao Tong University;* Xie Miao, Wei Du, and Chao Yu, *Huawei Technologies Co., Ltd.;* Haibo Chen, *Shanghai Jiao Tong University / Huawei Technologies Co., Ltd.*

QZFS: QAT Accelerated Compression in File System for Application Agnostic and Cost Efficient Data Storage . . . 163
Xiaokang Hu and Fuzong Wang, *Shanghai Jiao Tong University, Intel Asia-Pacific R&D Ltd.;* Weigang Li, *Intel Asia-Pacific R&D Ltd.;* Jian Li and Haibing Guan, *Shanghai Jiao Tong University*

Big-Data Programming Models & Frameworks

Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies 177
Youngseok Yang and Jeongyoon Eo, *Seoul National University;* Geon-Woo Kim, *Viva Republica;* Joo Yeon Kim, *Samsung Electronics;* Sanha Lee, *Naver Corp.;* Jangho Seo, Won Wook Song, and Byung-Gon Chun, *Seoul National University*

Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics 191
Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu, *The Chinese University of Hong Kong*

STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model 207
Jin Kyu Kim and Abutalib Aghayev, *Carnegie Mellon University;* Garth A. Gibson, *Carnegie Mellon University, Vector Institute, University of Toronto;* Eric P. Xing, *Petuum Inc, Carnegie Mellon University*

SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads 223
Ashraf Mahgoub, *Purdue University;* Paul Wood, *Johns Hopkins University;* Alexander Medoff, *Purdue University;* Subrata Mitra, *Adobe Research;* Folker Meyer, *Argonne National Lab;* Somali Chaterji and Saurabh Bagchi, *Purdue University*

Security #1: Kernel

libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK) 241
Soyeon Park, *Georgia Institute of Technology;* Sangho Lee, *Microsoft Research;* Wen Xu, *Georgia Institute of Technology;* Hyungon Moon, *Ulsan National Institute of Science and Technology;* Taesoo Kim, *Georgia Institute of Technology*

Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers 255
Jia-Ju Bai, *Tsinghua University;* Julia Lawall, *Sorbonne Université/Inria/LIP6;* Qiu-Liang Chen and Shi-Min Hu, *Tsinghua University*

LXDs: Towards Isolation of Kernel Subsystems 269
Vikram Narayanan, *University of California, Irvine;* Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, and Michael Quigley, *University of Utah;* Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev, *University of California, Irvine*

JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre 285
Nadav Amit, *VMware Research;* Fred Jacobs, *VMware;* Michael Wei, *VMware Research*

Parallelism & Synchronization

Multi-Queue Fair Queuing 301
Mohammad Hedayati, *University of Rochester;* Kai Shen, *Google;* Michael L. Scott, *University of Rochester;* Mike Marty, *Google*

BRAVO—Biased Locking for Reader-Writer Locks 315
Dave Dice and Alex Kogan, *Oracle Labs*

Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems 329
Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo, *Huazhong University of Science and Technology*

Programmable I/O Devices

NICA: An Infrastructure for Inline Acceleration of Network Applications 345
Haggai Eran, *Technion–Israel Institute of Technology & Mellanox Technologies;* Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein, *Technion–Israel Institute of Technology*

E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers 363
Ming Liu, *University of Washington;* Simon Peter, *The University of Texas at Austin;* Arvind Krishnamurthy, *University of Washington;* Phitchaya Mangpo Phothilimthana, *University of California, Berkeley*

INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive 379
Zhenyuan Ruan, Tong He, and Jason Cong, *UCLA*

Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval 395
Shengwen Liang and Ying Wang, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing; University of Chinese Academy of Sciences;* Youyou Lu and Zhe Yang, *Tsinghua University;* Huawei Li and Xiaowei Li, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing; University of Chinese Academy of Sciences*

Graph Processing Frameworks

SIMD-X: Programming and Processing of Graph Algorithms on GPUs 411
Hang Liu, *University of Massachusetts Lowell;* H. Howie Huang, *George Washington University*

LUMOS: Dependency-Driven Disk-based Graph Processing 429
Keval Vora, *Simon Fraser University*

NeuGraph: Parallel Deep Neural Network Computation on Large Graphs 443
Lingxiao Ma and Zhi Yang, *Peking University;* Youshan Miao, Jilong Xue, Ming Wu, and Lidong Zhou, *Microsoft Research;* Yafei Dai, *Peking University*

Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines 459
Eunjae Lee, *UNIST;* Junghyun Kim, *TmaxOS;* Keunhak Lim, *Nexon;* Sam H. Noh, *UNIST;* Jiwon Seo, *Hanyang University*

Virtualization Flavors

From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers 475
Sadjad Fouladi, Francisco Romero, Dan Iter, and Qian Li, *Stanford University;* Shuvo Chatterjee, *unaffiliated;* Christos Kozyrakis, Matei Zaharia, and Keith Winstein, *Stanford University*

Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries 489
Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, and Michael L. Scott, *University of Rochester;* Kai Shen and Mike Marty, *Google*

A Retargetable System-Level DBT Hypervisor 505
Tom Spink, Harry Wagstaff, and Björn Franke, *University of Edinburgh*

***MTS*: Bringing Multi-Tenancy to Virtual Networking** 521
Kashyap Thimmaraju and Saad Hermak, *Technische Universität Berlin;* Gabor Retvari, *BME HSNLab;* Stefan Schmid, *Faculty of Computer Science, University of Vienna*

Security #2: Isolation

StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone 537
Heejin Park and Shuang Zhai, *Purdue ECE;* Long Lu, *Northeastern University;* Felix Xiaozhu Lin, *Purdue ECE*

CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves 555
Meni Orenbach, *Technion;* Yan Michalevsky, *Anjuna Security;* Christof Fetzer, *TU Dresden;* Mark Silberstein, *Technion*

Secured Routines: Language-based Construction of Trusted Execution Environments 571
Adrien Ghosn, James R. Larus, and Edouard Bugnion, *EPFL*

Supporting Security Sensitive Tenants in a Bare-Metal Cloud 587
Amin Mosayyebzadeh, *Boston University;* Apoorve Mohan, *Northeastern University;* Sahil Tikale, *Boston University;* Mania Abdi, *Northeastern University;* Nabil Schear, *MIT Lincoln Laboratory;* Trammell Hudson, *Two Sigma;* Charles Munson, *MIT Lincoln Laboratory;* Larry Rudolph, *Two Sigma;* Gene Cooperman and Peter Desnoyers, *Northeastern University;* Orran Krieger, *Boston University*

Exotic Kernel Features

Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs 603
Gyusun Lee, Seokha Shin, and Wonsuk Song, *Sungkyunkwan University;* Tae Jun Ham and Jae W. Lee, *Seoul National University;* Jinkyu Jeong, *Sungkyunkwan University*

M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication 617
Nils Asmussen, Michael Roitzsch, and Hermann Härtig, *Technische Universität Dresden, Germany; Barkhausen Institut, Dresden, Germany*

(continued on next page)

Deduplication

SmartDedup: Optimizing Deduplication for Resource-constrained Devices **633**
Qirui Yang, Runyu Jin, and Ming Zhao, *Arizona State University*

Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! **647**
Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat, *Dell EMC*

Exotic Kernel Features #2

GAIA: An OS Page Cache for Heterogeneous Systems **661**
Tanya Brokhman, Pavel Lifshits, and Mark Silberstein, *Technion—Israel Institute of Technology*

Transkernel: Bridging Monolithic Kernels to Peripheral Cores **675**
Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin, *Purdue ECE*

Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame **693**
Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan, *University of Pennsylvania*

SemperOS: A Distributed Capability System **709**
Matthias Hille, *Technische Universität Dresden;* Nils Asmussen, *Technische Universität Dresden; Barkhausen Institut;* Pramod Bhatotia, *University of Edinburgh;* Hermann Härtig, *Technische Universität Dresden; Barkhausen Institut*

Key-Value Stores

Pragh: Locality-preserving Graph Traversal with Split Live Migration **723**
Xiating Xie, Xingda Wei, Rong Chen, and Haibo Chen, *Shanghai Jiao Tong University*

ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores **739**
Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu, *University of Science and Technology of China*

SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores **753**
Oana Balmau, Florin Dinu, and Willy Zwaenepoel, *University of Sydney;* Karan Gupta and Ravishankar Chandhiramoorthi, *Nutanix Inc.;* Diego Didona, *IBM Research–Zurich*

Unification of Temporary Storage in the NodeKernel Architecture **767**
Patrick Stuedi, *IBM Research;* Animesh Trivedi, *Vrije Universiteit;* Jonas Pfefferle, *IBM Research;* Ana Klimovic, *Stanford University;* Adrian Schuepbach and Bernard Metzler, *IBM Research*

Solid-State & Hard Disk Drives

Evaluating File System Reliability on Solid State Drives **783**
Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder, *University of Toronto*

Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. **799**
Jaeho Kim, *Virginia Tech;* Kwanghyun Lim, *Cornell University;* Youngdon Jung and Sungjin Lee, *DGIST;* Changwoo Min, *Virginia Tech;* Sam H. Noh, *UNIST*

Practical Erase Suspension for Modern Low-latency SSDs **813**
Shine Kim, *Seoul National University and Samsung Electronics;* Jonghyun Bae, *Seoul National University;* Hakbeom Jang, *Sungkyunkwan University;* Wenjing Jin and Jeonghun Gong, *Seoul National University;* Seungyeon Lee, *Samsung Electronics;* Tae Jun Ham and Jae W. Lee, *Seoul National University*

Track-based Translation Layers for Interlaced Magnetic Recording **821**
Mohammad Hossein Hajkazemi, *Northeastern University;* Ajay Narayan Kulkarni, *Seagate Technology;* Peter Desnoyers, *Northeastern University;* Timothy R Feldman, *Seagate Technology*

Networking

Your Coflow has Many Flows: Sampling them for Fun and Speed 833
Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin, *Purdue University*

PostMan: Rapidly Mitigating Bursty Traffic by Offloading Packet Processing 849
Panpan Jin, Jian Guo, and Yikai Xiao, *National Engineering Research Center for Big Data Technology and System, Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology, China;* Rong Shi, *The Ohio State University, USA;* Yipei Niu and Fangming Liu, *National Engineering Research Center for Big Data Technology and System, Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology, China;* Chen Qian, *University of California Santa Cruz, USA;* Yang Wang, *The Ohio State University, USA*

R2P2: Making RPCs first-class datacenter citizens 863
Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion, *EPFL*

Lancet: A self-correcting Latency Measuring Tool 881
Marios Kogias, *EPFL;* Stephen Mallon, *University of Sydney;* Edouard Bugnion, *EPFL*

Non-Volatile Memory

Pangolin: A Fault-Tolerant Persistent Memory Programming Library 897
Lu Zhang and Steven Swanson, *UC San Diego*

Pisces: A Scalable and Efficient Persistent Transactional Memory 913
Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen, *Shanghai Jiao Tong University*

Scheduling Things

EdgeWise: A Better Stream Processing Engine for the Edge 929
Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee, *Virginia Tech*

Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads 947
Myeongjae Jeon, *UNIST and Microsoft Research;* Shivaram Venkataraman, *University of Wisconsin and Microsoft Research;* Amar Phanishayee and Junjie Qian, *Microsoft Research;* Wencong Xiao, *Beihang University and Microsoft Research;* Fan Yang, *Microsoft Research*

Storage Failure & Recovery

Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures 961
Erci Xu, *Ohio State University;* Mai Zheng, *Iowa State University;* Feng Qin, *Ohio State University;* Yikang Xu and Jiesheng Wu, *Alibaba Group*

Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy 977
Amy Tai, *Princeton University and VMware Research;* Andrew Kryczka and Shobhit O. Kanaujia, *Facebook;* Kyle Jamieson and Michael J. Freedman, *Princeton University;* Asaf Cidon, *Columbia University*

Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems 993
Zhufan Wang and Guangyan Zhang, *Tsinghua University;* Yang Wang, *The Ohio State University;* Qinglin Yang, *Tsinghua University;* Jiaji Zhu, *Alibaba Cloud*

OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface 1009
Yun-Sheng Chang and Ren-Shuo Liu, *National Tsing Hua University*

Machine Learning Applications & System Aspects

Optimizing CNN Model Inference on CPUs 1025
Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang, *Amazon*

Accelerating Rule-matching Systems with Learned Rankers 1041
Zhao Lucis Li, *University of Science and Technology China;* Chieh-Jan Mike Liang and Wei Bai, *Microsoft Research;* Qiming Zheng, *Shanghai Jiao Tong University;* Yongqiang Xiong, *Microsoft Research;* Guangzhong Sun, *University of Science and Technology China*

(continued on next page)

MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving 1049
Chengliang Zhang, Minchen Yu, and Wei Wang, *Hong Kong University of Science and Technology;* Feng Yan, *University of Nevada, Reno*

Cross-dataset Time Series Anomaly Detection for Cloud Systems 1063
Xu Zhang, *Microsoft Research, Nanjing University;* Qingwei Lin, Yong Xu, and Si Qin, *Microsoft Research;* Hongyu Zhang, *The University of Newcastle;* Bo Qiao, *Microsoft Research;* Yingnong Dang, Xinsheng Yang, Qian Cheng, Murali Chintalapati, Youjiang Wu, and Ken Hsieh, *Microsoft;* Kaixin Sui, Xin Meng, Yaohai Xu, and Wenchi Zhang, *Microsoft Research;* Furao Shen, *Nanjing University;* Dongmei Zhang, *Microsoft Research*

Refreshing ATC – USENIX ATC '19 Program Co-Chairs Message

Dan Tsafrir
Technion – Israel Institute of Technology and VMware Research

Dahlia Malkhi
VMware Research and Calibra

1 Introduction

Welcome to ATC '19: the 2019 USENIX Annual Technical Conference. The scope of ATC covers all practical aspects related to systems software, and its goal is to improve and further the knowledge of computing systems of all scales, from small embedded mobile devices to large data centers, while emphasizing implementations and experimental results.

The ATC '19 program is the result of tremendous efforts by many in our community. We are most thankful to the authors who submitted their high-quality work and to the reviewers who undertook the challenging task of evaluating hundreds of submissions and providing constructive feedback to the authors. While working on creating the program, we have been repeatedly inspired by our reviewers' competence, experience, patience, and dedication. Thanks to their efforts, we are happy to report that the excellent program of ATC '19 achieves its aforementioned goal.

Briefly, we received 356 submissions and accepted 71 (19.9% acceptance rate) through a double-blind, two-rounds review process. The statistics that describe the submitted and accepted papers, along with the details of the review process, are summarized in Table 1 and are further discussed below.

This document is somewhat longer than is typical for a "message from the ATC program co-chairs". What motivated us to write this detailed report is the many changes that have been introduced to ATC this year, the reasoning underlying them, and the new things we have learned while working on creating the program. The potential target audience for this document is future chairs, or readers who wish to learn more about the process.

count	*description*
i. all submissions (short & full):	
356	submitted (458 registered)
29	violated format, given 24 hour to fix
2	rejected+withdrawn due to said format violations
2	withdrawn before review process ended
352	underwent the full review process
184	promoted to review round #2 (R2)
80	R2 submissions pre-rejected during online discussions
37	R2 submissions pre-accepted during online discussions
67	R2 submissions discussed at PC meeting (accepted 34)
71	accepted (19.9% acceptance ratio)
ii. short submissions:	
32	submitted
1	rejected+withdrawn due to format violations
8	promoted to R2
7	R2 submissions pre-rejected during online discussions
1	discussed at the meeting and accepted
1	full submission accepted as short
iii. committee & reviewing load:	
66	heavy weight PC members; 18–19 reviews per member
28	light weight PC members; 13 reviews per member
22	external review committee (ERC) members; 5 reviews
116	committee members
51	external reviewers; 1 review
iv. reviews:	
3–4	per submission in R1 (at least 2 by heavy members)
5–6	per submission in R2 (at least 4 by heavy members)
1,347	reviews in R1
405	reviews in R2
1,752	total, consisting of 1,097,815 words (6.7MB)
v. authors:	
1,695	all submissions (1,442 unique, with 409 affiliations)
384	accepted (361 unique, with 118 affiliations)

Table 1: *ATC '19 submissions and reviewing statistics.*

2 Changes

We have introduced some notable changes to ATC this year, primarily to meet higher reviewing standards used by other major systems conferences. We discuss these changes next.

2.1 Increased Number of Reviews

Top-tier system conferences typically employ a two-rounds reviewing process in which each submission receives at least three reviews in the first review round (R1), and then, if the submission is promoted to the second round (R2) based on its R1 reviews, it gets at least two additional reviews, amounting to at least five reviews per R2 submission.

In contrast, until this year, ATC R1 and R2 submissions received only two and four reviews, respectively. Upon investigation, we have learned that the decision to employ fewer reviews than other systems conferences has been made more

than a decade ago, by the program co-chairs of ATC 2008.

We and many others believe that making review-round promotion decisions based on only two R1 reviews is less informed, and hence leads to higher variability in the result. We further feel that the minimal number of R2 reviews should be similar to that of the other main system conferences, to allow for a better, more rigorous paper selection process. Consequently, this year, all submissions have indeed received at least three R1 reviews and at least five R2 reviews.

2.2 Double Blindness

Ever since ATC has been established, and until this year, the conference has employed a single-blind reviewing process, whereby reviewers see the names of the authors of the submissions that they review. While simplifying the review process, studies show that single-blindness might lead to bias against minorities and in favor of well-known authors and organizations. For example, Tomkins et al. show that

> *"Reviewers in the single-blind condition typically bid for 22% fewer papers and preferentially bid for papers from top universities and companies. Once papers are allocated to reviewers, single-blind reviewers are significantly more likely than their double-blind counterparts to recommend for acceptance papers from famous authors, top universities, and top companies. The estimated odds multipliers are tangible, at 1.63, 1.58, and 2.10, respectively."* [14]

Similarly, Goues et al. show that

> *"Reviewers with author information were 1.76x more likely to recommend acceptance of papers from famous authors, and 1.67x more likely to recommend acceptance of papers from top institutions. [...] When reviewers knew author identities, review scores for papers with male-first authors were 19% higher, and for papers with female-first authors 4% lower."* [4]

The latter study also shows that reviewers are usually unable to deanonymize authors of submissions by guessing, even if they believe themselves to be experts on a submission's topic.

Accordingly, major systems conferences (including SOSP, OSDI, ASPLOS, Eurosys, FAST, NSDI, and USENIX Security) employ a double-blind reviewing process by keeping author identities concealed from reviewers.

For ATC '19, we employed this policy as well, and we hope future chairs will continue to do so. The ATC '19 call for papers (CFP) requires authors to make a good faith attempt to anonymize their submissions by avoiding identifying themselves or their institution, either explicitly or by implication, e.g., through references, acknowledgments, online repositories that are part of the submission, or direct interaction with committee members. When authors cite their own studies, the CFP specifies two possibilities: either cite them as written by a third party (preferable), or as anonymized supplemental material uploaded to the HotCRP submission management system (most useful when the cited work is currently under review or awaiting publication). Prior publication as a technical report or in an online repository does not constitute a violation of anonymity.

2.3 Author Responses

Most premier systems conferences – e.g., OSDI, SOSP, ASPLOS, USENIX Security – give authors a few days to write a response to the reviews. The authors' response is known as "rebuttal", and it is optional. It allows authors to provide answers to specific questions raised by reviewers and, importantly, to correct factual errors or misunderstanding in the reviews. (It may *not* provide new results or reformulate the presentation.) Some researchers perceive rebuttals as essential for the reviewing process, to keep it fair and transparent [6], and some ACM SIGs encourage program chairs and steering committees of SIG-sponsored events to employ rebuttals, based on feedback from their members [13].

Therefore, for ATC '19, we chose to allow authors to rebut. Similarly to our past experiences in forming programs while serving in committees of conferences that employ rebuttals, our sense is that the author responses have contributed to the ATC '19 process. Primarily because they allowed the reviewers to make better informed decisions in certain cases. But also because they implicitly encouraged reviewers to write more accountable reviews and, importantly, to submit them on time so as to be visible during the authors response period; the latter allowed the online discussion period to start on time with all the required material available.

We used a 500-words soft limit on the size of the rebuttal; reviewers were not required to read more. The reviews were made visible to authors in the rebuttal period, during which reviewers were asked to avoid modifying them. After the rebuttal period ended, reviews became invisible to authors again, allowing reviewers to update them based on the rebuttal, the online discussions, and the program committee (PC) meeting.

2.4 Submission Chairs

The ever-increasing number of submissions to systems conferences (approaching 400 in the last two ATCs) makes it increasingly challenging for everyone involved to create a program. For example, it is challenging for reviewers to bid on hundreds of submissions so as to express review preferences. It is likewise challenging to arrange things such that the submission system accurately reflects conflicts associated with more than a hundred reviewers and an order of magnitude more authors (experience repeatedly shows that many conflicts are missing because reviewers and authors neglect

to declare all their conflicts). It is also challenging to manage a "dual track" PC meeting (where the PC is split between two rooms part of the time) in a manner that ensures that all committee members are found in the right room at the right time in order to discuss the submissions they have reviewed. Many other examples exist.

For this reason, we decided to formalize the role of a "submission chair" as part of the official organizers of ATC. The job of the submission chair is to help the program chair in accomplishing tasks such as those listed above by, for example: adding missing conflicts to HotCRP based on DBLP; helping reviewers' bidding by identifying the submissions that cite their papers and communicating this information to the reviewers; checking format violation in uploaded PDFs and communicating with authors to quickly fix those through reformatting and content deletion; helping to ensure that the quality of the reviews assignment is high (HotCRP assignments might be far from optimal); helping to make sure that per-submission administrative tasks are being carried out and progress is achieved, e.g., by following up on submissions that were not yet tagged as passing the "review sufficiency check"; helping in scheduling of the dual track meeting; and serving as scribes during the meeting while making sure the scheduling of PC members in rooms works as expected.

Submission chairs get admin privileges in the HotCRP system in order to carry out their duties. Their role, however, never requires them to make decisions that affect the outcome of the review process. For example, they do not steer online discussions. It is productive for the program chair and submission chair to be geographically located near each other, allowing them to physically meet when the need arises.

2.5 Extended Review Committee (ERC)

Most of the premier systems conferences, which must review a few hundreds of submissions, typically employ a light-heavy program committee model, where "light" PC members review fewer submissions but do not attend the PC meeting, whereas "heavy" members review more submissions and attend the meeting. This model is needed in order to decrease the high reviewing load of PC members, while keeping in mind that the number of people who can sit in one room and conduct a productive discussion is bounded.

Last year, unpredictably, ATC '18 received nearly a hundred additional submissions as compared to ATC '17 (377 submissions as compared to 283 submissions, respectively). To our knowledge, the PC of 2018 was the the first ATC PC to employ the light-heavy model. In previous years, all ATC PC members were "heavy", which was viable because the number of submissions was much lower, albeit, even so, past ATC-s reviewing load was sometimes in the range of 25–30 submissions per member. (Some of us were members of those PCs and still remember the pain.)

Our goal for this year was to ensure that the reviewing load of heavy members will not exceed 20 submissions. In parallel, USENIX instructed us to be prepared for an additional sizable increase in the number of submissions. Therefore, to be safe and have some flexibility, we decided to supplement the light-heavy model with an Extended Review Committee (ERC), consisting of members whose review load will be light: about 5 submissions per member.

Notably, due to the light reviewing load, ERC members were easy to draft regardless of their seniority: they typically accepted our invitation (which specified that the expected reviewing load will be 3–7). Additionally, more than a quarter of the ERC members were initially invited to serve as heavy or light members and opted for the lighter alternative instead of declining altogether.

Ultimately, having an ERC was a contributing factor that allowed us to assign four reviewers in R1 to most submissions (without increasing the load on light and heavy members beyond our planned upper bound). Having an initial assignment of four reviews proved to be invaluable when making R2 promotion decisions in the face of multiple late reviews, as three reviews were typically enough to confidently make the call. The ERC members additionally contributed by augmenting the expertise of our pool of reviewers.

2.6 No Abstract Submission Deadline

Last year, in their welcome message, the program co-chairs of ATC '18 stated that

> *"We required authors to submit abstracts a week before the paper submission in the hope of ensuring proper subject area coverage by the program committee and to get an idea of the reviewing load. This did not work. We had over 550 submitted abstracts, meaning almost 40% of the submissions were abandoned. In the end, requiring abstracts to be submitted early did not help with planning due to such a large number of abstracts that did not result in a submission"* [5].

To that we add that requiring committee members to indicate reviewing preferences before the submission deadline would be a waste of their valuable time, as they will inevitably bid on submissions that will not materialize. Stating review preferences given hundreds of finalized submissions is already time-consuming and challenging enough, and needlessly making this task even harder is counterproductive.

Bidding on registered abstracts that will not materialize into submissions would additionally negatively affect the quality of the review assignment, because committee members frequently stop bidding when they feel they have already placed "enough" bids on submissions.

Consequently, this year, we have to cancel the requirement to register abstracts in advance, and we eliminated the corresponding deadline.

2.7 Submission Deadline Closer to New Year

The date at which accept/reject notifications for ATC submissions are sent to authors is typically set by USENIX to around mid April.[1] Accordingly, since 2013, the submission deadline of ATC has been scheduled at the end of January or in early February, which thus far allowed the committee to complete the reviewing process in time to comply with a mid-April author notification date. This year, however, we set an earlier submission deadline: January 10, 2019.

Three issues necessitated this change. First, we needed additional time for the authors response period (Section 2.3) and for the "review sufficiency check" period that preceded it (described in Section 8). Second, as noted in Section 2.6, we had to allocate a few days following the deadline to allow reviewers to place bids on submissions indicating their review preferences; traditionally, such bidding took place before the submission deadline, as authors were required to register an abstract a week in advance.

The third issue that motivated an earlier deadline is the increased number of submissions. To cope with this increase, we allocated two weeks for online committee discussions scheduled before the PC meeting, in order to allow the committee to converge to a decision regarding as many submissions as possible—failing to do so would mean ending up with too many submissions to discuss at the meeting. The increased submission number also required allocating the week following the bidding period in order to assign reviews to members in a manner that would later allow us to reasonably conduct a dual track PC meeting (see details in Section 6).

Scheduling the submission deadline to occur soon after New Year may partially explain this year's somewhat smaller number of submissions as compared to last year: 377 vs. 356 in ATC '18 and ATC '19, respectively.

2.8 Uniform Shepherding

In the past, shepherding in ATC was not used by default. This approach reduces the load from both committee members and authors. A main drawback, however, is the increased likelihood that some of the issues that the reviewers expect authors to address in the camera-ready version remain unresolved.

The alternative approach, used by most of the premier systems conferences, is to assign shepherds to all accepted papers and thereby generally improve quality assurance. As part of our efforts to update the ATC reviewing process in order to make it aligned with that of its sibling conferences, this year, we decided that all accept decisions are conditional and depend on the approval of shepherds.

After the (conditional) accept notification, authors were given a few days to consider how to address the reviewers' comments and email a revision plan to their shepherd. Authors and shepherds then agreed on a timeline that allows the authors to complete the revision, providing enough time for the shepherd to read, consider, and discuss the revision with the authors, while permitting a final round of text polishing if necessary before the camera-ready deadline. At the end of this process, shepherds explicitly "signed off" the inclusion of papers in the program using HotCRP tags, allowing the program chairs to track the progress of turning all conditional accepts to accepts.

2.9 Accept as Short

As members of former ATC PCs, we are aware of full submissions that were accepted to past ATC-s on the condition that their authors will reduce their size to meet the short paper page-limit requirement. ATC program committees made such decisions rarely, limiting them to situations where the alternative is to otherwise reject the paper.

Surprisingly, past ATC call-for-papers were not clear about the possibility to accept as short; the practice was only anecdotally documented in the messages from chairs [2]. Seeing that this practice has been used in the past and may be used in the future, in the interest of transparency, we decided to explicitly declare it in the CFP, which now states that "the program committee may rarely decide to accept a full submission on the condition that it is cut down to fit in the short paper page limit" [19].

This CFP update initiated a discussion with USENIX board members who were concerned that the effort required to transform a full submission to a short paper might be too significant to accomplish between the authors notification date and the camera-ready date. They cited the FAST policy—which states that "the program committee will not accept a full paper on the condition that it is cut down to fit in the short paper page limit" [20]—as potentially preferable.

After consideration, we decided to keep the ATC accept-as-short policy because we believe it produces a significantly better outcome for both the authors and for the community, provided the alternative is to reject. In such rare cases, disallowing the PC to accept as short would result in a lose-lose situation: the authors lose because they are rejected instead of being given a chance to shorten and thereby get accepted; the ATC program loses a short paper; and the systems community loses because the paper would be subsequently resubmitted and hence re-reviewed, requiring the community to spend additional reviewing cycles, whereas reviewing load is already too high.

[1] In odd years, if the appropriate coordination takes place (as is the case this year), ATC notifications occur shortly before the SOSP submission deadline, to allow rejected authors of the former conference to submit an improved version of their study to the latter conference, assuming they have kept working on it while it was under submission at ATC.

2.10 Shorter Presentations

Last year's aforementioned 33% increase in the number of ATC submissions (377 in ATC '18 vs. 283 in ATC '17) and the consequent 27% increase in accepted papers (76 in ATC '18 vs. 60 in ATC '17) motivated the program co-chairs of ATC '18 to avoid hosting "best of the rest" sessions in their program, as well as to generate a longer-than-usual program that ends in the evening of the third day of the conference rather than around lunch time.

Despite having a similarly-sized program this year (71 papers), we wanted to have our cake and eat it too, namely: bring back the "best of the rest" sessions; further add lightning sessions to the program (see Section 2.12); while still end the program around lunch time at the third day, as was done in previous years prior to ATC '18.

To this end, this year, we decided to shorten the presentation time from 25 minutes per paper to 20 minutes. We believe that this change constitutes a reasonable compromise, allowing the conference to accommodate the additional sessions within the traditional time frame, while still providing enough time for presenters to convey the gist of their ideas.

2.11 Poster Requirement

To partially compensate for the shorter presentation time slots, this year, we dedicated the two poster sessions exclusively to accepted papers, and we required all paper-presenting authors to additionally present a poster in one of these sessions. Hopefully, this format will promote and facilitate interaction between authors and attendees who are interested in their work.

2.12 Lightning Sessions

In recent years "lightning sessions" have become standard in top-tier computer architecture conferences (ISCA, ASPLOS, etc.), and this year we decided to adopt them in ATC. Lightning sessions are typically interesting and fun, and, importantly, they are particularly suitable for conferences that have parallel sessions, which inevitably means attendees miss some of the presentations they are interested in. Lightning sessions give attendees a chance to make more informed decisions regarding what interests them the most and which talks are more worthy of their time. Speakers indeed often treat their lightning session presentations as previews aimed at soliciting listeners to attend the associated talks.

A lightning session is a joint session at the beginning of the day, which includes all the talks that will be given on that particular day. After the daily lightning session, the conference splits into its parallel tracks. Shortly before the daily lightning session, the speakers of that day queue in order—they do not sit until they present. Then, each lightning talk is allocated 120 seconds.

Each daily lightning session has a session chair. The chair is responsible for: interacting with speakers to get their slides beforehand; ordering slides on her laptop based on their order in the program, and making sure they display nicely; informing the speakers regarding the order; and regulating time during the session if necessary (we have never witnessed a lightning session chair having to actually exercise this authority).

Lightning speakers are additionally requested to submit lightning videos beforehand, which are made available in the conference web page before the conference. Both lightning presentations and videos are currently available in the ATC '19 technical sessions webpage.

In the past, USENIX conference talks were videoed, a very useful service that largely stopped due to financial reasons. Our hope is that lighting videos, which do not incur video recording costs, can partially provide some of this service: optimally, lightning videos would allow people who wish to only understand the gist of the idea to do so in 120 seconds.

3 Changes to Consider

3.1 Steering Committee

The one remaining notable difference between ATC and its sibling academic systems conferences (USENIX-sponsored: FAST, NSDI, OSDI, USENIX Security; SIGOPS-sponsored: ASPLOS, Eurosys, SOSP) is that ATC does not have a formal, broad, long-term steering committee. To make ATC more valuable to the community, we—nearly all ATC program chairs since 2015—believe that ATC should have such a committee, and we propose to form it, thus completing the transition of ATC into a conference that is governed by policies generally acceptable in the academic systems community.

We propose that the newly formed ATC steering committee will assume all responsibilities typically assigned to such committees, including providing advice and guidance to the current program co-chairs, selecting future program co-chairs, sustaining organizational memory, suggesting and considering new ideas when the need arises; and ultimately shaping the role of ATC. The identity of the steering committee members should be publicized along with call-for-papers to allow interested parties to address the committee with respect to matters that concern the conference long-term.

The members of the committee could, for example, be the USENIX executive director, relevant members of the USENIX board, and the program chairs from the last n ATC instances, such that members who chaired ATC in year $Y - n$ will be replaced by the ATC chairs of year Y shortly after the latter conference takes place. Joining the steering committee will of course be voluntary.

In October 9, 2018, a letter consisting of the content of this subsection has been submitted to the USENIX board. The letter was signed by all the ATC program chairs since 2015 except two (one responded too late and the other serves on

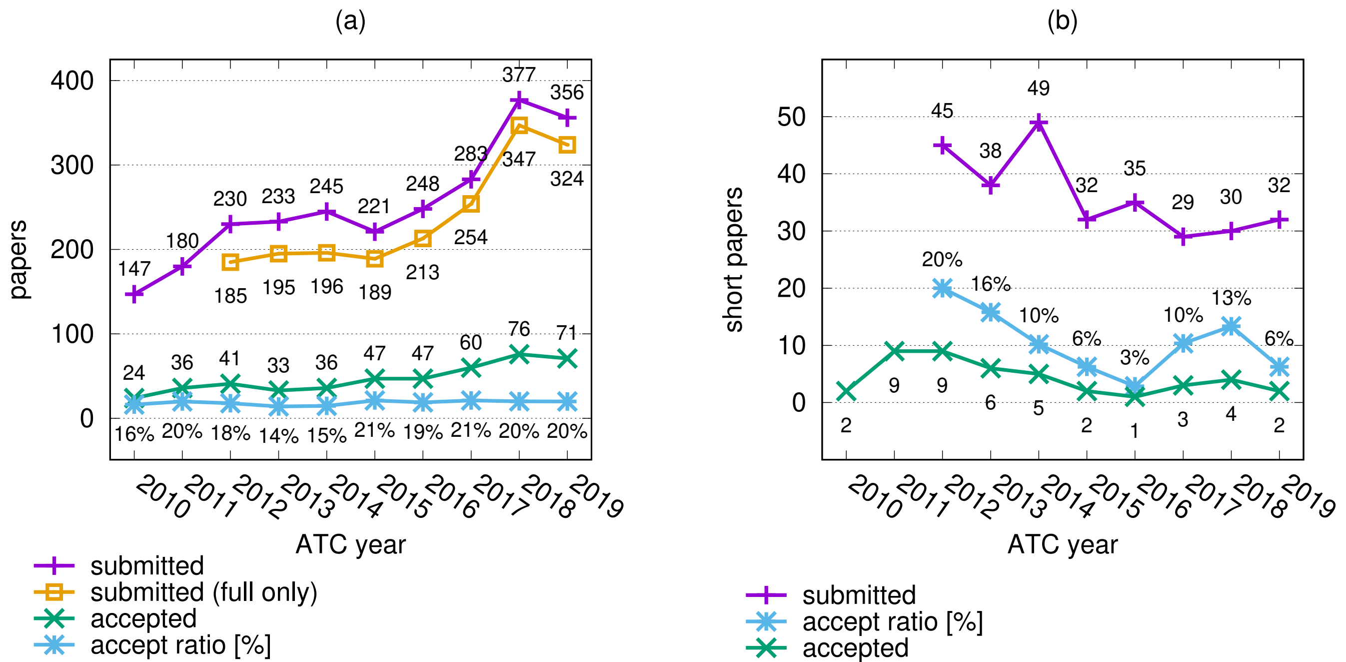


Figure 1: *(**a**) Submission and acceptance statistics of ATC papers (including both full and short) in the last decade, based on the corresponding proceedings' message from the ATC program chairs. (**b**) The same, but only for short papers. In 2010–2011, short submission numbers were not reported. In 2019 and 2013, one and three of the accepted short papers were submitted as full, respectively; we do not have this information for the other years.*

the Usenix board and is one of the decision makers regarding the steering committee issue). All who signed volunteered to serve on the steering committee when it is formed. The response of the relevant people in USENIX seems positive, but a steering committee has not yet been established.

3.2 Test of Time Award

All the premier systems conferences—except ATC—annually recognize historical, influential papers that have stood the test of time. This includes: USENIX Test of Time Award for FAST, NSDI, and USENIX Security [18]; SIGOPS Hall of Fame Award, which is typically handed to SOSP and OSDI papers [17]; Eurosys Test of Time Award [16]; and SIGARCH/SIGPLAN/SIGOPS ASPLOS Influential Paper Award [15].

The procedure to select the awarded papers varies. A common way employed is for the program committee of the conference to nominate influential papers published in that conference around ten years ago, with the final selection being made by the conference's steering committee (which, as noted, ATC still does not have). ATC is much older than ten years and, arguably, has changed its nature and goals over the years. So when/if an ATC test of time award is established, the steering committee will need to decide how to address older papers and handle the backlog. Jeff Mogul documented some of SIGOPS's considerations when establishing its Hall of Fame Award in 2005 and addressing similar questions [12].

3.3 Short Submissions

Figure 1a shows the submission and acceptance numbers of ATC papers in the last decade. Figure 1b shows the same statistics for short papers only. Getting a short submission accepted to ATC is clearly harder. We do not know why and can only speculate about the reason. Perhaps there is a mismatch between PC members' expectations and what can actually be accomplished in the scope of a short paper. Perhaps authors wrongfully believe that the bar for short submissions is lower. And perhaps there is a loose negative correlation between the increasing number of full submissions and the decreasing number of accepted short papers because PC members feel they have stronger papers to accept, relatively speaking.

Regardless of the reason, the fact that ATC PCs have reviewed 29–35 short submissions per year in the last five years only to accept 1–4 of them raises the question of whether the effort is worth it, since the reviewing effort to accept short papers is significantly greater than the effort to accept full papers (3%–13% vs. about 20% acceptance rate for short and short+full submissions in the last five years, respectively).

This year provides an extreme demonstration of how much harder the PC has to work in order to accept short papers. Table 2 specifies the number of reviews that the ATC '19 PC wrote for full and short submissions, as well as the resulting

scenario	*submission type*	*written reviews*	*accepted papers*	*work ratio*
real (worst case)	full	1620	70	23:1
	short	132	1	132:1
extrapolated (best case)	short	132	4	33:1

Table 2: *The number of reviews that the ATC '19 PC wrote for full and short submissions demonstrates that the PC had to work much harder in order to accept a single short paper ("real"). Even if we hypothetically assume that the PC had accepted four short paper instead of one as in last year (best case scenario in the last five years), the reviews-to-accepts work ratio would still be nearly 1.5x higher ("extrapolated").*

number of accepts. It turns out that the PC wrote 132 reviews in order to accept a single short paper, as opposed to writing "only" 23 reviews in order to accept a full submission. Namely, the PC had to work nearly 6x times as hard.

That said, as can be seen in Figure 1b, this year has been especially bad for short submissions. But even if we hypothetically assume the best case scenario across the last five years of accepting four short papers, the corresponding reviews-to-accepts ratio would have been 35:1, which is still nearly 1.5x harder than accepting a full paper.

ATC enjoys a steadily increasing number of full submissions. As a consequence, the reviewing load becomes heavier, requiring bigger PCs that already hardly fit into one room. Considering the relatively low return on investment (a significantly higher reviews-to-accepts ratio), it may make sense for future ATCs to consider to stop soliciting short papers.

We note in passing that, this year, we revised the CFP definition of short submissions to exclude workshop-style papers ("a short paper is not like a workshop paper—it presents a complete idea, which does not require full length to be appreciated" [19]). We introduced this change hoping to increase the short submission success rate by discouraging authors from submitting work that (our experience suggests) ATC reviewers tend to reject. The data shown in Figure 1b suggests this change was ineffective .

3.4 Early Rejects or R1 Rebuttals

The program co-chairs of this year debated about the issue of whether or not to send early reject notifications to authors of submissions who did not make it to R2. The reasoning to oppose sending early rejects was that such notifications might provide an unfair advantage to R1 rejects over R2 submissions that will be rejected later on, because the authors of the former will be free to resubmit their work elsewhere much sooner. Additionally, early rejects might translate to even higher reviewing loads that the community must handle due to said earlier resubmissions. Lastly, and importantly, postponing the R1 reject notification would allow PC members to re-calibrate during the second round and the deliberations and potentially change their opinion.

The reasoning to supported early rejects was that delaying reject notifications would be counterproductive for authors who do not abuse the system but rather leverage the reviewers' feedback to improve their work before they resubmit. Arguably, the ATC reviewing process should not replace one evil ("helping" authors who might abuse the system by ignoring the reviewers' feedback and resubmitting prematurely) with another (allowing authors to believe that they have a chance to get accepted for a good few weeks whereas in fact they do not).

Eventually, since we already introduced many changes to ATC this year (Section 2), we decided to leave things as they are in this particular case and avoid sending early reject notifications. But we encourage future ATC program chairs (and/or the ATC steering committee if it is established) to reconsider.

Because decisions were collectively sent to authors shortly after the PC meeting, R1 rejects were given a chance to write a rebuttal (Section 2.3), which the committee members read and considered. Two R1 rejected submissions were resurrected as a result. These submissions were promoted to R2 and urgently assigned two additional reviewers. In the end, however, both were rejected. We speculate that allowing authors to rebut (also) after R1 (as is done by some conferences) would have had a bigger effect. But doing so would require more labor and an even earlier deadline, which would be closer to New Year, which might result in fewer submissions (see Section 2.7).

3.5 Physical PC Meeting

The number of submissions the PC can discuss in one day (let us denote it as c) is bounded. For example, it takes more than eight hours to discuss $c = 70$ submissions if allocating 7 minutes per submission, as is typical. PCs also usually dedicate 2–3 minutes to present each submission that was pre-accepted in the online discussion phase (ATC '19 had 37 such submissions), and they take about 30 minutes for lunch. It is challenging to squeeze all these activities into one day.

Let m denote a member of the PC, and let r denote the number of submissions reviewed by m. Similarly to c, the value of r is bounded. At the risk of overgeneralizing, we roughly approximate that $r = 15$, $r = 20$, and $r = 25$ reviews per member are nowadays considered light, average, and heavy reviewing loads in academic systems conferences, respectively. The value of r cannot be raised arbitrarily.

In contrast to c and r, the total number of submissions that the PC must review (let us denote it as n) is unbounded and keeps increasing. The practical meaning of this increase is that, on average, fewer and fewer of the r submissions that m reviewed are getting discussed at the meeting. Figure 2 demonstrates this trend, assuming $c = 70$ submissions are discussed at the meeting, and that 2/3 and 1/3 of the r submissions assigned to m are reviewed in R1 and R2, respectively. The x axis shows n, and the y axis shows the corresponding

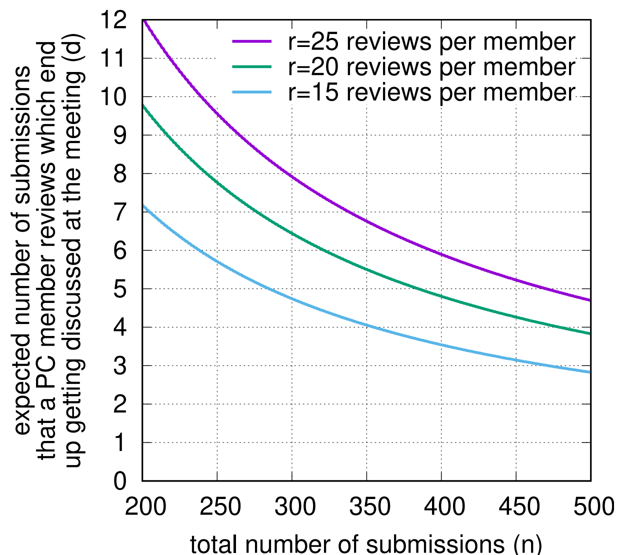


Figure 2: *Increased number of submissions translates to fewer submissions that each PC member gets to discuss at the PC meeting; see Appendix A for details.*

expected number of submissions that have been reviewed by m and ended up being discussed at the meeting (let us denote it as d), which is monotonically decreasing.

In Appendix A, we show that under our assumptions, $d \approx 4rc/3n$ is a reasonable approximation of the expected number of submissions that m reviewed and discussed at the meeting. As noted, because r and c (numerator) are bounded, d asymptotically behaves like $1/n$ (denominator).

Our PCs received $n = 356$ submissions and used an upper bound of 18–19 reviews per heavy member, which more or less corresponds to the line associated with $r = 20$ in Figure 2. In the relevant range of n, we see that $d = 5.5$ submissions discussed at the meeting per member. Because d is just an average, some members discussed more submissions, but others discussed less: as little as 2–3 submissions in certain cases. Flying to California to discuss such a small number of submissions is, arguably, counterproductive.

In 2018, the PC meeting spanned across two days, allowing the committee to make fewer decisions during the online discussions period and instead discuss $c = 124$ submissions in person at the meeting (with $n = 377$ and $r = 18$). Therefore, by our calculation, each member discussed about 8 submissions on average, alleviating the problem somewhat. On the other hand, 8 submissions during two days means 4 submissions per day (as compared to 5.5 per day in 2019), which is not necessarily preferable.

When discussing this issue with some of the members during the PC dinner, it seemed like most agreed that there is a problem: the time overhead and carbon emission associated with physical PC meetings are possibly becoming excessive considering the smaller number of submissions that each member gets to discuss. Still, there was a sense that the program turned out better due to the physical meeting, which allowed the members to calibrate. Additionally, several members—both junior and senior—pointed out that a notable value they get from PC meetings is the chance to network and interact with their peers.

In light of the above, it may be advisable for future program chairs to consider if in-person, physical PC meetings are worth it, at least in their current format. If they decide in favor of physical meetings, one conceivable way to increase their value is, for example, to couple them with workshop-style events, where committee members briefly present their ideas and get feedback from their peers.

4 Assembling the Committee

After we accepted the position of the ATC '19 program co-chairs, we were asked by USENIX to take into account that the number of submissions in 2019 might exhibit the same growth rate as it did in 2018, which would bring us to about 500 submissions (a.k.a. "the nightmare scenario" :-)), requiring $3 \times 500 + 2 \times 250 = 2000$ reviews assuming 50% of the submissions move to R2 (see Section 2.1). A smaller, more conservative estimate of 400 submissions would require $3 \times 400 + 2 \times 200 = 1600$ reviews. In comparison, a sizable heavy PC of 60 members each contributing 20 reviews—a threshold we were hoping and planning not to exceed—provides $60 \times 20 = 1200$ reviews. Taking into account these numbers, we decided to draft a heavy PC, a light PC, and an ERC (see Section 2.5) with target sizes of 65, 25, and 25, respectively.

Drafting about 115 committee members is a challenging task. In preparation for it, we compiled a list of all those who served on PCs in the last three instances of the main systems conferences, such that we had a pool of candidates to helps us (we used: ASPLOS 2017–2019, ATC 2016–2018, Eurosys 2017–2019, FAST 2017–2019, NSDI 2017–2019, OSDI/SOSP 2016–2018, and USENIX Security 2016–2018).

Analyzing this database brought up an interesting insight, which might indicate that our community has scalability issues in terms carrying out the reviewing load. Table 3 shows the relevant statistics. The aggregated sum of the size of the 21 PCs we have included in our analysis is 1118. These membership positions were manned by 655 unique individuals, a finding that could be interpreted to mean that members serve in $1118/655 \approx 1.7$ PCs in three years, on average. A deeper look at the data, however, reveals that 284 individuals participated in two or more of the PCs in our database, and these individuals are responsible for manning 783 (70%) of the 1118 positions. This finding implies that a relatively small group of people shoulders most of the reviewing load.

Figure 3 depicts the histogram of how many of the members in our database (y) served in how many of the PCs that we included (x), which demonstrates the reviewing effort dis-

memberships (aggregated sum of PC sizes)	1,118
number of unique members	655
number of unique recurring members	284

Table 3: *Membership statistics of the PCs of the main systems conferences in the last three years.*

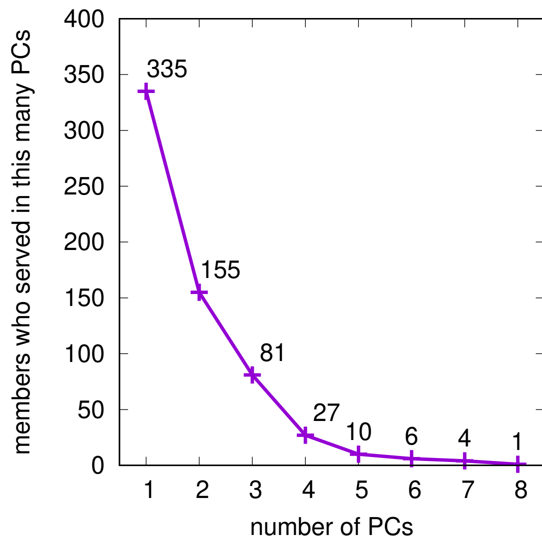


Figure 3: *Histogram showing how many of the members of the PCs of the main systems conferences in the last three years (y axis) served in how many of these PCs (x axis).*

invite type	invite sent	accepted as heavy	accepted as light	accepted as ERC	declined
heavy	131	66	16	3	46
light	22	-	12	3	7
ERC	27	-	-	16	11
sum	180	66	28	22	64

Table 4: *Number of invitations to serve on the ATC '19 committee sent to candidates, and the corresponding responses.*

seniority	junior	31
	senior	63
gender	female	14
	male	80
sector	university	64
	industry	25
	both	5
continent	N. America	69
	Europe	15
	Asia	5
	Middle East	4
	Australia	1

country	USA	62
	Canada	7
	Switzerland	6
	Israel	4
	UK	4
	Germany	2
	Netherlands	2
	Korea	2
	Australia	1
	China	1
	France	1
	Hong Kong	1
	Sweden	1

Table 5: *Demographic information of the PC (heavy and light, excluding program co-chairs).*

parity. We can see, for example, that one member served in 8 PCs, and four members served in 7.

The list we compiled was helpful in drafting the committee. When sending heavy member invitations, we allowed the candidates to accept as light or ERC, and when sending light member invitations, we allowed the candidates to accept as ERC. The number, type, and outcome of the invitations are specified in Table 4, and the demographic information of the resulting PC is specified in Table 5. Nearly 2/3 of the invitations sent were accepted, and as can be seen, this relatively high success rate is partially because we allowed candidates to opt for roles that involve a smaller reviewing load.

5 Conflicts and Bidding

5.1 Missing Conflicts

Before assigning submissions to reviewers, it is important for the submission management system, HotCRP, to have accurate conflict of interest information as defined by the ATC '19 call for papers [19]. In addition to the conflict information that authors and reviewers explicitly specify, HotCRP helps by highlighting potential conflicts based on the information available to it, which is productive. This year, we also used the PC Chair Kit [3] that was written for ISCA '18 to find missing conflicts based on authorship information available via DBLP.[2] The script downloads the relevant DBLP information and checks if there are any co-authors of submission authors from the last n years that are not already listed as HotCRP conflicts.

Our submission co-chairs found 150 such undeclared conflicts and verified them manually. They identified a few false positives (e.g., two researchers with identical name, a summer school report authored by many authors that should not be considered as a real conflict), but the rest of the conflicts were valid.

5.2 Helping Committee Members to Bid

Authors associate topics from a predetermined list with their submissions, and committee members declare their per-topic level of (dis)interest for each such topic. This information is important, because it is utilized by HotCRP to compute a per-member score for each submission, and members use these scores to sort through hundreds of submissions and thereby ease the process of bidding—the act of associating integers with submissions to indicate reviewing preference. HotCRP then uses bids (as well as topic scores when, e.g., bids are absent) to assign reviews to reviewers.

Instructions for Committee Members We requested committee members to favor bidding on submissions for which

[2] More accurately, we used a fork of that kit [7].

they can provide expert or knowledgeable reviews, rather than on submissions that they find interesting but do not fall in their area of expertise.

We additionally requested committee members to limit the range of the numeric values they use to express preference to -20 to 20. The HotCRP system does not compare preference values of different users in the automatic review assignment algorithm and so members need not use the same scale. Some review assignments, however, are inevitably done manually by program chairs, and then having a common scale is helpful.

Defining Topics Last year, in ATC 2018, the aforementioned predetermined list consisted of 62 topics, as opposed to years 2017 and 2016, at which ATC used a list consisting of 17 topics. Some speculate that having this many topics is cumbersome, overly verbose, and unhelpful [9], and we seriously considered minimizing the list and consolidating topics when defining it for 2019. But a closer look at the historical data (from ATC '18, as well as from ASPLOS '19, which used a similarly sized list) indicated that authors and reviewers do use most topics in the longer lists.

Considering that (1) the task of bidding is really hard when there are hundreds of submissions, and that (2) PC members do primarily rely on topics when bidding as a way to cope with this submission volume, we eventually decided that it might be counterproductive to shrink the topic list and risk making bidding harder. A concise (or at least coarser grained) list could be preferable, and mining past data more seriously may provide evidence that support this hypothesis. But as we currently do not know, we decided to stick with the more sizable, finer grained list (although we made changes).

Figure 4 shows the 59 topics used in ATC '19, ranked by the number of submissions that used them. It could be argued that even our least popular topic ("cryptography", which was associated with only three submissions) is worthwhile, because it is preferable for the associated submissions to be reviewed by the appropriate committee members who are actually capable of doing it, and it seems reasonable to speculate that the odds of that happening would have been smaller without the topic.

Grouping Topics Given that there are dozens of topics, it makes sense to group related topics when they are presented to authors and committee members within HotCRP, which makes using them easier. In ATC '18, the program co-chairs did so in an ad hoc manner by adding grouping prefixes to topic strings that are separated from the topic names by a colon (for example: "storage:deduplication", "storage:disk (CMR, SMR, etc.)", "storage:erasure coding", and so on). In ATC '19, we used the same notation but also kindly requested the HotCRP maintainer to directly support the concept, which he did [9], making the HotCRP presentation of grouped topics more elegant, usable, and effective. The topic groups we used are: general, devices, networking, OS, PL/SE (abbreviation of programming languages and software engineering), security, storage, systems, and techniques/aspects.

total number of citations of committee papers	1266
average number of citations per member	11.6
median number of citations per member	7
standard deviation	11.5
citations of top-most cited member	67
citations of 2nd-most cited member	61
citations of 3rd-most cited member	43

Table 6: *Statistics of citations of committee member papers found in the ATC '19 submissions and communicated to members to help with their bidding.*

Pinpointing Submissions that Cite Members As noted, having to place bids to decide which submissions to review is becoming more challenging due to the increasing number of submissions. Merely reading the titles of 300–400 submissions is time-consuming, and many reviewers need more information than just the title to decide to bid. Attempting to ease the process of bidding, we generated for, and shared with each committee member a list that specifies all the ATC '19 submissions that cite that member's papers. The list was generated by our submission co-chairs using the aforementioned PC Chair Kit [7].

Table 6 provides some statistics about the citations we have found. Since there are more than a thousand of them, hopefully, they provided a usable signal to some of the committee members.

Dealing with Unpopular Submissions Despite the fact that nearly 90% of the committee members placed positive bids on 20 submissions or more (and 2/3 of the members placed positive bids on 40 submissions or more), some submissions were associated with relatively few positive bidders. Perhaps unsurprisingly, some submissions are much more popular than others. The line associated with "before" in Figure 5 depicts the disparity of popularity. The x axis shows the rank of each submission based on the the number of members that bade positively on it, and y axis shows the corresponding number of bids.

Focusing on the bottom right, we can see that 60 submissions received only 6 positive bids or less, which would have likely hampered the review assignment process. We therefore labeled these 60 as "lowbids" in HotCRP and asked our committee members to consider positively bidding on some of them if they are within their domain of expertise, stating that if everyone does this truthfully, no one will be tasked with arbitrary submission assignments. The line associated with "after" in Figure 5 demonstrates that this request was effective. (Albeit the data is distorted somewhat by the fact that the "after" line additionally accounts for bids we solicited before the beginning of R2.)

Figure 4: *The ATC '19 topics (without their grouping prefix) ranked by the number of submissions that used them.*

Figure 5: *Disparity of popularity among ATC '19 submissions.*

Interestingly, out of those 60 "unpopular" submissions, only three (5%) were accepted to the ATC '19 program, which is 4x lower than the overall acceptance rate. Perhaps this poor success rate suggests that bidding information could be leveraged somehow to make the reviewing process more efficient? A positive answer to this question would be helpful, because the ATC '19 committee wrote 281 reviews for these particular 60 submissions, which is a lot of effort in order to accept only three.

On the other hand, one of these three has been awarded best paper, which is another demonstration of what all of us already know: popularity isn't everything... :)

6 Planning for a Dual-Track PC Meeting

Due to the increasing number of submissions to system conferences, in order to be able to finish the PC meeting on time, several recent program chairs resorted to splitting the meeting into two parallel tracks for part of the time, such that each track is simultaneously headed by a different co-chair. Assuming that the number of submissions is not going down any time soon, it seems like dual-track meetings are here to stay.

However, properly organizing a dual-track meeting is challenging. Notably because it may affect how submissions are assigned to reviewers, as it is nontrivial to arranges things such that *all* PC members are *always* found in the right room at the right time while the meeting takes place.

Currently, there is no standard, generally accepted best-practice for how to arrange a successful dual-tack PC meeting. Program chairs typically need to apply creativity and to spend much effort to come up with an appropriate model they feel would work and would be suitable for their committee. For this reason, before we describe the model we used, we survey the models of dual-track meetings used by program chairs before us, and we briefly discuss their pros and cons. Hopefully, this discussion would be useful for future chairs when deciding upon the model that works best for them, as the state of the art of dual-track PC meetings evolves.

ASPLOS '17 Model The first PC that we are aware of that split into tracks occurred spontaneously at the PC meeting of ASPLOS '17, when attending members and chairs realized it was not realistic for them to finish on time. They therefore split in an ad hoc manner to flexible, parallel discussion groups. The approach was reported to have worked: the program was ready at the end of the day, and the members lived to tell the tale.

ASPLOS '18 Model In the subsequent year, having experienced the difficulties from the previous year, the program co-chairs of ASPLOS '18 carefully planned for the dual-track meeting. They split their PC members into two disjoint equally-sized sets M_0 and M_1, with the stated goal of having equal expertise in both, in all the relevant conference topics. They likewise split the submissions into two equally sized

sets S_0 and S_1, and they exclusively assigned submissions from S_i to M_i, such that no PC member reviewed outside of her sub-committee's pool of submissions. Consequently, by design, running the dual track meeting was easy.

A main concern with this model is that it splits the expertise and thus runs the risk of arbitrarily preventing the most appropriate experts who happen to belong to M_i from reviewing submissions that happen to belong to the "wrong" pool $S_{(i+1) \bmod 2}$.

ASPLOS '19 Model In an effort to alleviate this drawback, the program co-chairs of ASPLOS '19 employed the following approach in deciding how to define M_i and S_i. ASPLOS is an interdisciplinary venue of three communities: SIGARCH (50% sponsorship), SIGOPS (25% sponsorship), and SIGPLAN (25% sponsorship). Accordingly, the chairs initially divided their PC into M_{OS} and M_{PL} containing members from the operating systems community and the programming languages community, respectively. They then searched for an "optimal" division of the PC members from the architecture community into two parts, each added to the initial M_{OS} and M_{PL} to form two equally-sized M_{OS}^{arch} and M_{PL}^{arch} sets that, together, comprise the entire PC.

The said optimality was achieved as follows. The chairs and their helpers used a script that exhaustively enumerated all the possible equally-sized M_{OS}^{arch} and M_{PL}^{arch} group partitions. For each partition, they assigned every submission to the group that maximizes the submission's "affinity" (a combination of reviewer citations, topic score, and normalized bids). Then, they scored that partition by aggregating the affinity across all submissions within their assigned group. The final partition was the one that scored the highest by this metric.

They then calculated the "partitioning penalty" for each submission, which is the total affinity of the submission for the whole PC minus its affinity to the group it was assigned to. They assigned high partitioning penalty papers to the whole PC, thus adding a requirement for a *joint session* at the meeting, in addition to the dual track. To make workload for the two groups even, they took the most highly penalized papers from the larger group and assigned them to the whole PC.

The ASPLOS '19 model is more careful in how it splits S_i and M_i as compared to the ASPLOS '18 model, trying to minimize the penalty associated with splitting. It additionally supports submissions that are discussed jointly. Still, while minimized, the penalties do exist.

We note in passing that the ASPLOS '19 program co-chairs received extensive help in planning for their dual-track meeting from individual whose role was similar to what we formalized as "submission chairs" (Section 2.4).

ATC '18 Model The program co-chairs of ATC '18 decided not to split the PC beforehand and globally assign reviews across all members without any constraints. This approach is simple and entirely eliminates the penalties of splitting. The cost, however, is shifting all the administrative complexity to the PC meeting itself: it raises the question of how to run the dual-track meeting without resorting to the ASPLOS '17 model, which seems to have heavily relied on luck.

The ATC '18 program co-chairs did not rely on luck. They were successful in planning the dual-track PC meeting after (1) all the reviews have been uploaded, (2) the online discussions have been concluded, (3) the list of submissions to be discussed at the meeting have been finalized, and (4) it became known which PC members will call-in rather than attend physically.

The PC meeting timeline was divided into several consecutive sessions T_i $(i = 1, 2, ...)$, such that in each session T_i the PC was split into two groups T_i^i and T_i^{ii} that met in parallel. The group membership changed across sessions, so group T_1^i was different than group T_2^i, for example.

In some sessions, groups T_i^i and T_i^{ii} were disjoint. But in other sessions, some PC members were instructed to physically move to the other group at some point, but such transitions were limited to one move per one member per session. In such non-disjoint sessions, PC members were asked to be aware of the discussion schedule so as to know when to make the transition. But inevitably this did not always work, and so occasionally members were called from the other room. Still, the program co-chairs reported that, overall, the movement between rooms was minimal and not distracting.

One ATC '18 co-chair concluded that "if I would repeat, I would not change what [we] did because it worked fine, and the PC didn't seem to be bothered to move around." But the other co-chair reported that "I would avoid doing what we did in the future even though it worked amazingly well. We lucked out [...], and we barely pulled it off."

Similarly to ASPLOS '19, the ATC '19 program co-chairs received extensive help in scheduling the PC meeting from individuals whose role was similar to what we formalized as submission chairs.

ATC '19 Model Like the program co-chairs of ATC '18, we wanted to refrain from the penalties and complexities involved in splitting the PC beforehand in a manner that affects how reviews are assigned. But we also wanted to completely avoid the aforementioned transitions between rooms, the occasional missing members that must be fetched from elsewhere, and—perhaps most importantly—the sense of uncertainty associated with the "barely pulled it off" sentiment quoted above. We achieved all these goals as described next.

Immediately after the submission deadline passed, the committee members placed their bids, and missing conflicts were identified and uploaded, we repeatedly applied the following simulation procedure.

1. Using standard HotCRP functionality, simulate assigning three R1 reviewers to all submissions as if for real.

2. Randomly select 50% of these submissions (177 in our case) to be the simulated R2 submissions; let us denote this random set as S_2.

3. Using HotCRP functionality yet again, simulate assigning two additional R2 reviews by heavy members to all the submissions in S_2.

4. Randomly select 50% of the S_2 submissions (88 in our case) to be the simulated set of submissions to be discussed at the meeting; denote this random set as S_3.

5. Using a constraint solver, find a split of the heavy PC into two groups that allow for the longest simulated dual-track parallel session of submissions from S_3 (without any transitions of members between the two groups); submissions that cannot be discussed in parallel in this split, will be discussed in a simulated joint session.

6. Compute the time it takes to run these simulated parallel and joint sessions, assuming a 6–7 minutes discussion per submission. If the simulated meeting takes less than eight hours, declare success; otherwise declare failure.

Our submission co-chairs repeated the above procedure multiple times using multiple random selections, and they verified that it *always* declared success. We therefore gained confidence that scheduling our dual-track meeting using a constraint solver is doable, despite using a global review assignment. This was indeed the case in the actual PC meeting.

Before running the above experiment, we did not know whether or not it would be successful, and we were prepared to get a negative result. In this case, we planned to use the framework we developed to attempt to understand the root cause of the failure, and to try to devise constraints for the baseline HotCRP review assignment algorithm that would resolve the underlying issue. Thankfully, we did not have to do that.

HotCRP Multi Live-Meeting Trackers HotCRP has a useful live meeting tracker feature, which helps program chairs run the meeting by keeping attendees in sync, presenting the current and next submissions discussed and the relevant conflicts. The problem was that HotCRP assumed a single track meeting, making the tracker unusable in the case of dual tracks. Thankfully, again, the HotCRP maintainer was willing to accommodate our request to add support for multiple live-meeting trackers [10], which we indeed used in our meeting.

7 Review Assignment Improvements

The review assignment is done by HotCRP using a min-cost max-flow algorithm [8, 11]. This assignment utilizes member bids and topic scores in order to distribute the reviews among reviewers in a manner that attempts to be balanced and fair, both in terms of number of reviews assigned to each member, and in terms of the bidding preferences, such that everyone would hopefully get as many of their top bids as possible.

The review assignment process of the individual conferences frequently involves some constraints that must be taken into account when the assignment takes place. In the case of the first review round of ATC '19, these were: (1) each PC member gets an assignment of 13 reviews; (2) each ERC member gets an assignment of 5 reviews; and (3) each submission gets at least 2, and at most 3, reviews by heavy members.

There is no way we are aware of to express multiple constraints such as these all at once in HotCRP (nor in the underlying min-cost max-flow algorithm, we believe). Instead, a sequence of assignments is conducted that is applied to the various types of members: first heavy, then light, then ERC, and some creativity is involved to get the desired outcome, which is an assignment that adheres to all the constraints.

With the goal of checking the quality of the resulting assignment, we have defined the per-reviewer "goodness" metric as follows. Let n be the number of reviews assigned to the reviewer, namely, in our case, n is 13 and 5 for PC and ERC members, respectively. The goodness metric measures how many of the reviewer's most-preferred n submissions, associated with her highest bid values, were actually assigned to that reviewer. For example, if an ERC member was assigned her five most preferred submissions, then her goodness is $5/5 = 100\%$, but if she was assigned only one of them, then her goodness is $1/5 = 20\%$.

The line that approaches 0% in the bottom right of Figure 6 shows the goodness produced by the default HotCRP assignment algorithm for all PC/ERC members. The committee members are ranked based on their review goodness value, from highest to lowest, and this rank is displayed along the x axis; the y axis shows the goodness value of the corresponding members. The drop towards zero at the right indicates that the default algorithm might produce an unfair assignment when used as described above. Some members get all their top picks and some get none, with 31 members (more than 1/4 of the committee) members getting less than 60% of their top picks. Moreover, the default algorithm made 38 and 6 assignments where the bid placed by the corresponding members was zero or negative, respectively.

For these reasons, we implemented a script that helps improve the assignment as follows. Let r_i be a reviewer, s_i be some submission that r_i was assigned to review, and $b(r_i, s_i)$ be the numeric bid value that r_i placed on s_i. Our script initially attempts to exploit the fact that the default algorithm does not produce a stable marriage [21]. Namely, it is possible to find a subset of n reviewers r_i $(i = 0, 1, ..., n)$, each assigned with a certain submission s_i, such that if r_i hands s_i to $r_{(i+1) \bmod n}$ and reviews $s_{(i-1) \bmod n}$ instead, then: (i) no conflict of interest is violated; (ii) $b(r_i, s_i) \leq b(r_i, s_{(i-1) \bmod n})$, namely, the new assignment is at least as good as the previ-

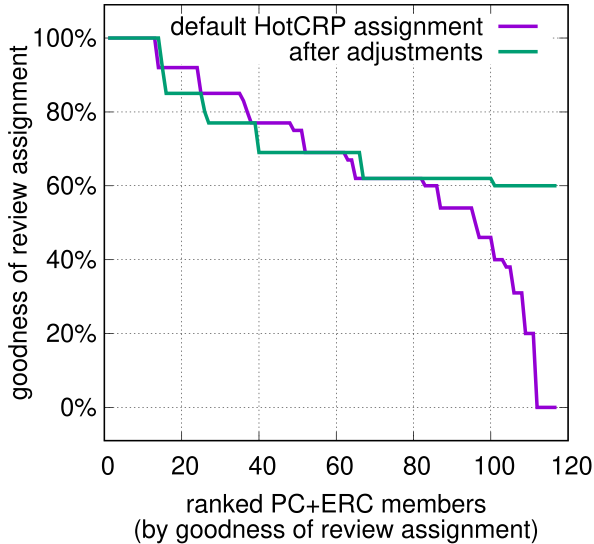


Figure 6: *Per-member goodness of the default HotCRP review assignment, which we improved, obtaining a lower bound of 60% through (i) review swaps that improved the assignments for all reviewers involved, or (ii) at the expense of reviewers who enjoy a much higher goodness value.*

ous for all reviewers involved; (iii) there exist at least one k ($0 \leq k < n$) for which $b(r_i, s_i) < b(r_i, s_{(i-1) \bmod n})$, namely, the new assignment is better than the old for at least one reviewer; and (iv) each submission still gets at least two and at most three heavy reviewers.

The script is repeatedly applied to the member currently associated with the lowest goodness value, who assumes the role of r_k defined in constraint (iii). The script attempts to find a submission switch as defined above, using $n = 2$ and $n = 3$. If no such swap exist, the script relaxes constraint (ii) so as to tolerate goodness reductions due to the swap, provided that the reviewers that suffer the reduction still enjoy a high goodness value after the switch.

Our script initiated 748 HotCRP events to adjust the original default assignment, as specified in Table 7. In the end, as shown in Figure 6, we were able to ensure a minimal goodness value of 60% to all members (namely, PC members got at least 8 of their top-13 preferences assigned to them, and ERC members got at least 3 of their top-5). Additionally, we were able to arrange things such that all committee members were exclusively assigned submissions associated with their positive bids, with two types of rare exceptions: (1) reviewers whose number of positive bids was smaller than 13 for PC or smaller than 5 for ERC; and (2) submissions with only one positive bid by a heavy PC member. In the latter case, the heavy member with the highest topic score was assigned as the second heavy reviewer.

Processing of the review assignment for R2 was similar albeit somewhat more challenging to improve, due to having

HotCRP events	*purpose*
215	eliminate assignments with zero or negative bids
12	at most 3 heavy reviewers per submission
494	increase low goodness to promote fairness
748	sum

Table 7: *Number of individual HotCRP events affecting review assignment that were generated by our script to improve upon the default assignment of R1.*

fewer usable bids, because only heavy members were assigned reviews, and also because of the additional constraint that we could only assign submissions to members who did not yet review them in R1.

Out of the 5–6 additional R2 reviews assigned to heavy members, the initial HotCRP review assignment assigned about 1/4 of the members with 1–5 submissions with which they associated a zero or negative bid. Anecdotally, one such member started off with *all* of his assignments having negative bids. Subsequently, we were able to adjust things such that all committee members were assigned submissions that are exclusively associated with their positive bids, with a few exceptions similar to those found in R1. Overall, half of the heavy PC members were assigned at least three of their (remaining) top picks, and all the them were assigned at least two of their top picks.

8 Reviewing Process

We employed a double-blind reviewing process consisting of two rounds, and we followed standard procedures for handling conflicts of interest. The PC consisted of 66 heavy and 28 light members, assisted by 22 ERC members. Additionally, 51 external reviewers contributed when specific expertise was required. The committee members were allowed to submit papers to the conference; the program co-chairs and submission co-chairs avoided it.

Table 1 summarizes the reviewing process. Out of 458 HotCRP registrations, we received a total of 356 submissions, divided into 324 full submissions (11 pages plus references) and 32 short submissions (5 pages plus references).

Format Violations We visually inspected all the submitted PDFs as well as used the HotCRP style checker to identify 29 submissions that violated the formatting rules. These were given a day to rectify the problem without making any content modifications; if fixing increased the size beyond the page limit, authors were required to remove (never change) content to meet the limit. All violating submissions complied except two, which were then rejected and withdrawn by the co-chairs.

Round 1 In Review Round 1 (R1), the PC members mostly contributed 13 reviews, and the ERC members mostly con-

tributed 5 reviews. Out of all R1 submissions, 277 were assigned four reviewers, and 75 were assigned three reviewers. Regardless, all of the submissions were assigned at least two reviews by heavy members (typical), and at most three. The committee wrote a total of 1,347 R1 reviews.

Round 2 We promoted 184 submissions to Review Round 2 (R2). We assigned each R2 submission with two additional reviewers from the heavy PC. A submission was promoted to R2: (i) if two or more reviewers gave it a positive score ("weak accept" or above); (ii) if a single positive reviewer decided that she supports promotion after considering the other reviews and despite of them, and, if she has so chosen, discussing the matter with the other, negative reviewers; or (iii) if the submission had fewer than three reviews due to late members.

To qualify to be the aforementioned "single positive reviewer", a member must have assigned a score of "accept" or "strong accept". For submissions with three (rather than four) reviews, a "weak accept" also qualified, provided the associated expertise was at least "knowledgeable" or the confidence was "high". Out of the 40 single-supporter submissions (24 with one "accept" or higher), we promoted 17 to R2 (13 with "accept" or higher). The committee wrote 405 R2 reviews and a total of 1,752 reviews in the two review rounds.

Review Sufficiency Check A few days before the rebuttal period, we applied a Review Sufficiency Check (RSC) procedure to all R2 submissions, to ensure that the reviews provide sufficient feedback to authors, as well as sufficient information to the committee to make an informed decision regarding the submission. To this end, for each R2 submission, we appointed one of the reviewers who is a heavy PC member as the "lead" of the submission. Leads were responsible for conducting the RSC by: (1) reading all the associated reviews; (2) asking the relevant reviewers to revise their reviews when the need arises (e.g., by calling out subjective claims that a submission is incremental without adequate citations of prior work, by identifying unclear statements, etc.); and (3) deciding together with the other reviewers if additional reviews are needed when expertise is low.

Online Discussions After the authors uploaded their rebuttals, we discussed the submissions online. Our goal until the meeting was to: (1) revise reviews if needed due to rebuttals; (2) revive R1 submissions if their rebuttals justify it (this happened in only two cases); (3) discuss submissions and attempt to reach consensus, color-tagging them as red to indicate preliminary reject, green to indicate preliminary accept, and yellow to indicate that reviewers are unable to reach consensus, so the submission should be discussed at the meeting; and (4) for red submissions that have a rebuttal, as well as for green submissions, write a post-discussion summary comment, which will be made visible to authors after the PC meeting, briefly explaining the primary reasons for rejections and possibly ways to improve (red), or what is required for the camera-ready (green). Such a summary was eventually written for all submissions that uploaded a rebuttal.

Reviewers who changed their mind about a submission due to the rebuttal or to the other reviews were asked to consider adding a "post-rebuttal feedback" section to their review and explain why. (We requested not to make substantive changes to reviews outside this section, as the reviews have already been seen by the authors and so any changes need to be clearly identified and justified.)

All the submissions, including R1, were assigned discussions leads, whose job was to drive discussion, write the summaries, and ensure progress. We asked leads to make an honest effort to ensure that the opinions of non-heavy reviewers were adequately voiced and represented at the meeting. Non-heavy members were warmly encouraged to champion submissions that they believe should be accepted, and all reviewers were encouraged not to feel pressured to adopt a common denominator point of view, and not to hesitate to go against the majority. Reviewers were encouraged to reflect on each others' opinions, e.g., by considering previous work or confirming an opinion from an expert.

We asked the reviewers to stay positive when possible (particularly when it comes to out-of-the-box ideas) and to keep in mind that we should be looking for reasons to accept a paper rather than reject.

When reviewers were unable to reach consensus (yellow), the online discussion was expected to reconcile as many differences among the reviewers as possible, leaving only a few substantive differences for a focused PC meeting discussion. Namely, tagging yellow was not used as a way to procrastinate or reduce work, because it is impossible to discuss all R2 submissions in one day. The meeting was planned to be dedicated primarily to those submissions that actually require it, focusing on differences that the reviewers had already identified as important.

When making decisions, we requested reviewers to assume shepherding but not for adding new results. (All accepted papers were indeed assigned shepherds, responsible for making sure that revision expectations are met.) Of the R2 submissions, we pre-rejected 80, pre-accepted 37, and tagged 67 as yellow to discuss at the meeting.

During the online discussions, we recognized that about a dozen R2 submissions might not have reviews with enough expertise, so we urgently solicited additional reviews from relevant experts after the rebuttal period. In these cases, we emailed the authors and allowed them to rebut the additional review(s), copy-pasting their response as a comment in the HotCRP relevant page.

Program Committee Meeting The PC meeting took place between 8am–6pm, 12 April 2019, in the VMware campus in

Palo Alto, CA. The program co-chairs, submission co-chairs, and 60 heavy PC members attended the meeting in person, five called in, and one could not participate. The meeting consisted of a morning joint session (8am–12pm), a split session in two rooms (12:30pm–3pm), and an afternoon joint session (3:15pm–6pm), followed by a lively PC dinner.

The split session composition was determined with the help of a constraint solver as described in Section 6. The partition was completely disjoint, and no members transitioned between rooms while it took place. We discussed 12 green (preliminary accept) and 25 yellow (discuss) submissions in the morning joint session, and 7 green and 12 yellow submissions in the afternoon joint session. In the split session, one group discussed 8 green and 16 yellow submissions, and the other group discussed 10 green and 14 yellow submissions. We allocated 3 and 7 minutes discussion time for each green and yellow submissions, respectively.

Out of the 67 yellow submissions discussed, the PC accepted 34, which, together with the 37 preliminary accepts, resulted in a program of 71 papers, of which 2 are short. Accept decisions were reached by consensus, except in two cases that required a PC vote.

9 Best Paper Selection

The best paper award selection process proceeded in two phases. In the first phase, we combined several signals. One was an explicit ranking by reviewers marking papers worthy of consideration for best-paper; any paper marked for such consideration by two or more PC members was passed to the second phase. Additionally, we considered general review ranks and deliberations (both online and during the PC meeting), moving several additional top-ranking papers to the second phase. Last, we collected explicit nominations by PC members for the best paper award.

At the end of the first phase, we generated a short-list of eight papers. At this stage, we appointed a swat team of six PC members consisting of senior and experienced members of the systems research community. During a period of four weeks, the team read papers, and we deliberated each one separately for best-paper worthiness. Conflicted members were excluded from discussions of the relevant papers. We did not place a quota on the number of best-paper awards. Generally, the committee favored papers with original or surprising contribution, and/or ones that would spark interest and establish a new direction for follow on works.

At the end of the second stage, we elected three papers to receive best-paper awards for USENIX ATC '19.

Acknowledgments

The ATC '19 program is the result of the efforts of many. We thank the authors for submitting their work, and the committee members and external reviewers for working so hard to review the submissions. We are deeply indebted to our awesome submission co-chairs, Lalith Suresh and Gerd Zellweger, and also to Igor Smolyar, who helped whenever needed. We also thank the Lightning Talks co-chairs, Deniz Altinbuken and Aasheesh Kolli, and the Best of the Rest co-chairs, Amy Tai and Chia-Che Tsai. We thank Erez Zadok for managing submissions for which both program co-chairs were conflicted.

We thank Eddie Kohler for authoring and maintaining HotCRP, and for supporting us and promptly adding the features we needed. We thank the program co-chairs of ATC '18, Haryadi Gunawi and Benjamin Reed, for being responsive and providing lots of useful information. We are grateful to Emmett Witchel, who co-chaired ASPLOS '19, went through everything a few months before us, and served as a source of much needed knowledge and emotional support. We thank Sarita Adve for accurately documenting her excellent review process in ASPLOS '14 [1], which was quite helpful. We thank Emery Berger for suggesting the idea of test of time award for ATC, and Vijay Chidambaram for so nicely articulating the case for double blindness—much of the text in Section 2.2 originated from him. We also thank Or Hershkovitz for reviewing the math in Appendix A and for finding and elegantly fixing a bug.

We thank the USENIX staff for their outstanding conference management, and notably Casey Henderson, Hakim Weatherspoon, and Angela Demke Brown for their thoughtful advice and guidance; Angela and Hakim additionally reviewed this document (in very short notice), and they provided valuable and much appreciated feedback that helped us improve it.

Lastly, we thank VMware for sponsoring and hosting the PC meeting (and for paying for drinks at the PC dinner), and Sandra Barreto, Lori Blonn, and Sean Crotty for helping to organize the meeting.

References

[1] Sarita Adve. ASPLOS '14: Program chair's message. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages iv–ix, 2014. https://dl.acm.org/citation.cfm?id=2541940.

[2] Andrew Birrell, , and Emin Gün Sirer. Message from the 2013 USENIX Annual Technical Conference program co-chairs. In *ATC '13: USENIX Annual Technical Conference*, page vi, 2013. https://www.usenix.org/sites/default/files/atc13_message.pdf.

[3] Mario Drumond, Mark Sutherland, and Babak Falsafi. PC chair kit. https://github.com/mdrumond/pc-chair-kit.

[4] C. Le Goues, Y. Brun, S. Apel, E. Berger, S. Khurshid, and Y. Smaragdakis. Effectiveness of anonymization in double-blind review. *Communications of the ACM (CACM)*, 61(6):30–33, May 2018. http://doi.org/10.1145/3208157.

[5] Haryadi Gunawi and Benjamin Reed. Message from the 2018 USENIX Annual Technical Conference program co-chairs. https://www.usenix.org/sites/default/files/atc18_message.pdf, 2018.

[6] Gernot Heiser. Peer review: Anonymity should not be at the expense of transparency. https://microkerneldude.wordpress.com/2015/02/13/peer-review-anonymity-should-not-be-at-the-expense-of-transparency/, Feb 2015. Accessed: Jul 2019.

[7] Tyler Hunt. Fork of PC chair kit. https://github.com/tylershunt.

[8] Samir Khuller and Richard Matthew McCutchen. Assigning papers to reviewers. https://mattmccutchen.net/match/index.html, 2013.

[9] Feature request: ability to group topics. GitHub HotCRP issue https://github.com/kohler/hotcrp/issues/153, Dec 2018.

[10] Feature request: live-meeting-tracker for dual-track meetings. GitHub HotCRP issue https://github.com/kohler/hotcrp/issues/154, Dec 2018.

[11] Eddie Kohler. HotCRP source file mincostmaxflow.php. https://github.com/kohler/hotcrp/blob/master/lib/mincostmaxflow.php.

[12] Jeffrey C. Mogul. Policies for the SIGOPS hall of fame award. *SIGOPS Operating Systems Review*, 42(3):132–135, Apr 2008. https://doi.org/10.1145/1368506.1368525.

[13] Guidelines for the program chair of a SIGPLAN event. https://www.sigplan.org/Resources/Guidelines/ProChair/.

[14] Andrew Tomkins, Min Zhang, and William D. Heavlin. Reviewer bias in single- versus double-blind peer review. *Proceedings of the National Academy of Sciences (PNAS)*, 114(48):12708–12713, 2017. https://doi.org/10.1073/pnas.1707323114.

[15] SIGARCH/SIGPLAN/SIGOPS ASPLOS influential paper award. https://www.acm.org/sig-awards.

n	number of submissions submitted to the conference
n_2	number of submissions promoted to R2
m	an individual PC member
r	number of reviews written by m
r_1	number of reviews written by m in R1
r_2	number of reviews written by m in R2
c	number of submissions discussed at the PC meeting
d	reviewed by m and discussed at the PC meeting

Table 8: *Notation.*

[16] Eurosys test-of-time award. http://www.eurosys.org/awards/tot-10-award.

[17] SIGOPS – the hall of fame award. https://www.sigops.org/awards/hof.

[18] USENIX test of time awards. https://www.usenix.org/conferences/test-of-time-awards.

[19] USENIX ATC '19 call for papers. https://www.usenix.org/conference/atc19/call-for-papers.

[20] FAST '19 call for papers. https://www.usenix.org/conference/fast19/call-for-papers.

[21] Wikipedia. Stable marriage problem. https://en.wikipedia.org/wiki/Stable_marriage_problem.

Appendix A Submissions Discussed by Each Member at the Meeting

Let n denote the number of papers that have been submitted to the conference. Let n_2 denoted the total number of R2 submissions that have been promoted from R1. Let m denote one PC member, and assume that m has reviewed exactly r submissions out of the n. Further assume that the number of m's R1 and R2 reviews are r_1 and r_2, respectively ($r = r_1 + r_2$). Let c be the total number of submissions that have been *discussed* at the PC meeting, and let d denote how many of these c submissions have been reviewed by m ($d \leq r$). These notations are summarized in Table 8. Recall that Figure 2 shows that as n grows, d decreases, to the point that m has little to do at the PC meeting because d is small. The computation underlying Figure 2 assumes a typical setup for systems conferences where $n_2 = n/2$ (half of the submissions have been promoted to R2), $r_1 = \frac{2}{3} \cdot r$ and $r_2 = \frac{1}{3} \cdot r$ (two thirds of m's reviews are written during R1), and the number of discussed submissions is $c = 70$. With our assumptions, an intuitive approximation of d on average is

$$d \approx r_1 \cdot \frac{c}{n} + r_2 \cdot \frac{c}{n_2} = (r_1 + 2r_2) \cdot \frac{c}{n} = \frac{4rc}{3n} \quad (1)$$

because (1) the probability that a single R1 submission that has been reviewed by m will be discussed at the meeting is

c/n, and, similarly, (2) the probability that a single R2 submission that has been reviewed by m will be discussed is approximately c/n_2, if disregarding the fact that the latter probability is in fact affected by the specific number of R1 submissions reviewed by m that have made it into R2. (For example, if all the submissions that m reviewed in R1 were promoted to R2, then the latter probability should actually be $\frac{c}{n_2-r_1}$, seeing that m cannot be assigned R2-submissions that she has already reviewed in R1.)
Figure 2, however, does not depict the approximation of d but rather computes it accurately, as follows. Let $p(n,c,r_1,k)$ denote the probability that exactly k of the r_1 submissions that m reviewed in R1 have been discussed at the meeting, then

$$p(n,c,r_1,k) = \binom{r_1}{k} \cdot \binom{n-r_1}{c-k} \div \binom{n}{c}. \tag{2}$$

Thus, $e(n,c,r_1)$, which is the expected number of submissions that m reviewed in R1 and were discussed at the meeting, can (also) be computed with the following summation

$$e(n,c,r_1) = \sum_{k=0}^{r_1} p(n,c,r_1,k) \cdot k. \tag{3}$$

Now, by using Equations 2–3 and the law of total probability, we can compute $e_2(n,c,r_1,r_2)$, which is the expected number of submissions that m reviewed in R2 and were discussed at the meeting, as follows

$$e_2(n,c,r_1,r_2) = \sum_{k=0}^{r_1} p(n,n_2,r_1,k) \cdot e(n_2-k,c,r_2). \tag{4}$$

Notice that Equation 4 uses $p(n,\mathbf{n_2},r_1,k)$ instead of the earlier $p(n,\mathbf{c},r_1,k)$, because here the probability corresponds to the event that k of the r_1 submissions reviewed by m in R1 were promoted to R2. Using Equations 3–4, we conclude that

$$d = e(n,c,r_1) + e_2(n,c,r_1,r_2), \tag{5}$$

which allows us to compute d accurately instead of approximating it. That said, in the range plotted in Figure 2, the difference between the real value of d (Equation 5) and its approximation (Equation 1) is always smaller than 0.52, which is reasonably close.

The Design and Operation of CloudLab

Dmitry Duplyakin *Robert Ricci* *Aleksander Maricq* *Gary Wong* *Jonathon Duerig*
Eric Eide *Leigh Stoller* *Mike Hibler* *David Johnson* *Kirk Webb*
Aditya Akella[*] *Kuangching Wang*[†] *Glenn Ricart*[‡] *Larry Landweber*[*] *Chip Elliott*[§]
Michael Zink[¶] *Emmanuel Cecchet*[¶] *Snigdhaswin Kar*[†] *Prabodh Mishra*[†]

University of Utah [*]University of Wisconsin [†]Clemson University
[‡]US Ignite [§]Raytheon [¶]UMass Amherst

Given the highly empirical nature of research in cloud computing, networked systems, and related fields, testbeds play an important role in the research ecosystem. In this paper, we cover one such facility, CloudLab, which supports systems research by providing raw access to programmable hardware, enabling research at large scales, and creating a shared platform for repeatable research.

We present our experiences designing CloudLab and operating it for four years, serving nearly 4,000 users who have run over 79,000 experiments on 2,250 servers, switches, and other pieces of datacenter equipment. From this experience, we draw lessons organized around two themes. The first set comes from analysis of data regarding the use of CloudLab: how users interact with it, what they use it for, and the implications for facility design and operation. Our second set of lessons comes from looking at the ways that algorithms used "under the hood," such as resource allocation, have important—and sometimes unexpected—effects on user experience and behavior. These lessons can be of value to the designers and operators of IaaS facilities in general, systems testbeds in particular, and users who have a stake in understanding how these systems are built.

1 Introduction

CloudLab [31] is a testbed for research and education in cloud computing. It provides more control, visibility, and performance isolation than a typical cloud environment, enabling it to support work on cloud architectures, distributed systems, and applications. Initially deployed in 2014, CloudLab is now heavily used by the research community, supporting nearly 4,000 users who have worked on 750 projects and run over 79,000 experiments.

On the surface, CloudLab acts like a provider of cloud computing resources: users request on-demand resources, configure them with software stacks of their choice, and perform experiments. Much like a cloud, the testbed simplifies many of the procedures surrounding access to resources, including selection of hardware configuration, creation of custom images, automation for software installation and configuration, and more. CloudLab staff take care of the construction, maintenance, operation, etc. of the facility, letting users focus on their research. CloudLab gives the benefits of economies of scale and provides a common environment for repeatability.

CloudLab differs significantly from a cloud, however, in that its goal is not only to allow users to build applications, but entire clouds, from the "bare metal" up. To do so, it must give users unmediated "raw" access to hardware. It places great importance on the ability to run fully observable and repeatable experiments. As a result, users are provided with the means not only to *use* but also to *see*, *instrument*, *monitor*, and *modify* all levels of investigated cloud stacks and applications, including virtualization, networking, storage, and management abstractions. Because of this focus on low-level access, CloudLab has been able to support a range of research that cannot be conducted on traditional clouds.

As we have operated CloudLab, we have found that, to a greater extent than expected, "behind the scenes" algorithms have had a profound impact on how the facility is used and what it can be used for. CloudLab runs a number of unique, custom-built services that support this vision and keep the testbed operational. This includes a resource mapper, constraint system, scheduler, and provisioner, among others. CloudLab has had to make several trade-offs between general-purpose algorithms that continue to work well as the system evolves, and more tailored ones that provide a smoother user experience. The right choices for many of these trade-offs were not apparent during the design of the facility, and required experience from the operation of the facility to resolve.

The primary goal of this paper is to provide the architects of large, complex facilities (not only testbeds, but other IaaS-type facilities as well) with lessons from CloudLab's design choices and operational experiences. CloudLab is one of many facilities that serve the research community in various capacities [8, 6, 16, 33, 21, 34, 31, 35], and we aim to generalize the lessons from this specific facility. As a secondary goal, we hope that users of these facilities benefit from a closer look into the way they are designed and operated. With these goals in mind, this paper makes two contributions:

- In Section 2, we describe the CloudLab facility as it

has been built and analyze its basic usage patterns and the research conducted on it. This analysis, and the dataset that goes with it, represent a contribution to the **community understanding of the practical operation of IaaS-type facilities**.

- In Section 3, we analyze specific design choices using data from the operational system, looking at some of the trade-offs inherent in the facility's design. This analysis yields **important insights about how these choices affect user behavior and point to design principles for other facilities.**

Sections 4 and 5 cover related work and conclude.

2 Development and Use of CloudLab

We begin with background on CloudLab; our goal is not a complete summary of its goals, design, and deployment, but to provide sufficient context for the analyses that follow. We then examine usage patterns: how the use of the facility has evolved over time, the availability of resources, and the types of research that are conducted on it. From these analyses, we draw lessons about user behavior and look at the implications for the design of testbeds and IaaS facilities in general.

2.1 The Deployed CloudLab Facility

The primary CloudLab hardware is hosted at three sites: the University of Utah, Clemson University, and the University of Wisconsin–Madison. Though every site supports a wide variety of hardware-agnostic experimentation, each site specializes in a different area of research. Wisconsin's hardware is designed for networking and storage work, Clemson's for analytics and high-performance workloads, and Utah's for scale-out workloads. This equipment has come online in batches as CloudLab has been built out and evolved in response to user demand. Identical nodes in the same batch are all labeled with the same *hardware type* to help users request nodes with specific properties and to enable experiments to be repeated on the same types of resources. Since its initial public availability in December 2014, CloudLab has added devices such as programmable Ethernet switches, GPUs, Infiniband, and high-disk-count servers in response to user feedback. A full description of CloudLab's hardware can be found in its manual [36].

In addition to the hardware that it owns, CloudLab is *federated* [30, 7] with several other facilities, including Emulab [39] and Apt [32]. This brings the total number of servers available to CloudLab users up to about 2,250, and for the rest of the paper we include these resources in our analysis and discussion of CloudLab's hardware.

CloudLab is operated using software developed in-house specifically for running research testbeds: its control software is directly descended from software developed for the Emulab [39], GENI [25, 32], and Apt [32] testbeds. We have extended this software to better support experimentation on clouds and have made a number of improvements (such as those documented in Section 3) based on our experience running the facility.

CloudLab provides access to its devices at the *lowest layer* possible with a *minimum* of virtualization and abstraction between users and hardware. The reason for this is twofold. First, CloudLab's goal is to support research that is not possible on public (or typical private) clouds: it allows users to modify aspects of the software stack that would be fixed on those platforms, such as the storage, virtualization, and networking layers. Second, this supports more repeatable experimentation than facilities that virtualize and share their resources, as it provides strong performance isolation between tenants, factoring out the unpredictable "background noise" that makes it harder to draw sound, scientific conclusions. CloudLab takes pains to ensure that all servers of the same hardware type have comparable performance: in prior work [22], we have developed techniques for identifying servers whose performance is not statistically representative of the whole, and we exclude such servers from the population seen by experimenters. The facility takes the principle of low-level access beyond just servers and also provides "raw" access to other types of hardware such as programmable Ethernet switches [37] and servers with many drives from which users can build their own SANs.

Experiments in CloudLab are instances of *profiles*. A profile contains a description of the hardware resources (servers, switches, etc.) that the experiment will run on, and the software needed to run the experiment (in the form of disk images, `git` repositories, and scripts to run). When a profile is *instantiated*, CloudLab selects available hardware that matches the profile's specification and provisions that hardware with the software and configuration options described in the profile. Every instance of the profile runs on a separate set of hardware resources, and many instances of the same profile can run simultaneously. The CloudLab operators provide standard profiles for popular cloud software stacks, such as OpenStack [28], as well as bare-metal profiles that load standard Linux distributions. Users can also create their own profiles, which they can share with others. A typical workflow for creating a new profile involves starting with CloudLab-provided disk images, installing custom software, and creating a hardware description meeting the experiment's needs. All experiments have an *expiration*: when they are first created, they are set to expire after a few hours. Users can then request that their experiments be *extended* to last longer; short extensions (hours to days) are granted automatically (assuming resources do not need to be reclaimed to satisfy reservations), and administrators evaluate requests for longer periods (weeks to months). When deciding whether to grant these requests, administrators look at coarse-gained *idleness* statistics, such as CPU load and network packet counts, to de-

termine whether the user is using resources efficiently; other than this, CloudLab does not collect information about use *inside* of experiments. It is typical for there to be 200–300 experiments active on CloudLab at any point in time.

It is possible to fully script the workloads that run inside of an experiment, but in practice, most research done on CloudLab involves a great deal of development time and exploratory experiments, so most use is interactive. A key difference between CloudLab and typical cloud (as well as research and academic cyberinfrastructures such as Jetstream [33], Chameleon Cloud [21], and the Mass Open Cloud [35]) is that clouds place emphasis on *elasticity*, and therefore tends to treat ensembles of VMs working together as an *orchestration* problem. CloudLab's profiles place the emphasis instead on describing a *complete, repeatable environment*. This makes it less elastic, but makes it easier to describe entire networks and to repeat experiments in a consistent environment. We have found that some users do have initial confusion regarding this different focus, but that they tend to find it an easier way to run repeated experiments in the long run.

2.2 Hardware Overview

CloudLab Utah has a large number of servers, each with relatively modest specifications. 585 of the servers use HPE's high-density Moonshot platform, which places 45 low-power servers (Intel Xeon-D or ARM64 SoC) in each chassis. Each chassis contains two 10 Gbps switches, which effectively function as "top of rack" switches and are interconnected at 160 Gbps through a core switch. Another 200 servers connect to both a traditional Ethernet network (at 25 Gbps) and to a "layer-1" network. The latter allows control of the physical-layer topology, configurably "wiring" nodes to user-controllable Ethernet switches or directly to each other. These user-controllable Ethernet switches are allocated to one user at a time, allowing users to have full control over their configuration and even, in some cases, to program them.

CloudLab Wisconsin's goal is to reflect the type of technology and architecture found in a typical modern enterprise datacenter. All servers (which come from Cisco) are dual-socket and have a mix of spinning hard drives (HDDs) and solid state drives (SSDs). Several servers have large numbers of disks (up to 14), allowing users to build their own SAN configurations. Many are equipped with GPUs, enabling work on machine learning and applications of GPU computing to other areas, including network packet processing and other systems tasks. The network is arranged in a Clos topology.

CloudLab Clemson focuses on putting more CPU cores in each server (from Dell) and on a greater amount of RAM per core. This makes it suitable for hosting big data analytics (such as Hadoop and Spark), for running high-performance computing workloads, and for hosting large numbers of virtual machines. The Ethernet experiment network topology uses three interconnected core switches, each connected to a companion top-of-rack switch handling direct server connections. In addition to its Ethernet network, CloudLab Clemson has a 40 Gbps QDR Infiniband network used for HPC and RDMA experiments.

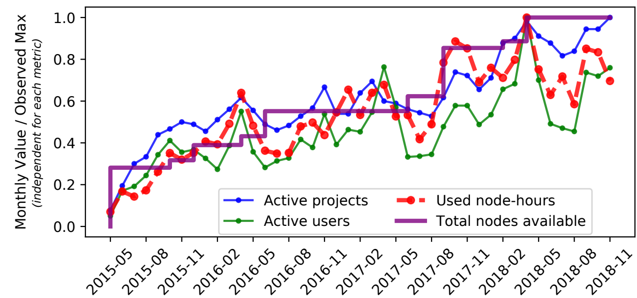


Figure 1: Growing testbed capacity and utilization. To produce a consistent scale, we divide monthly values by the all-time maximum value for each metric.

The hardware at each site has grown over time: the number of servers has increased approximately fourfold over the period covered in this paper. No hardware has yet been retired.

2.3 Usage Patterns

Figure 1 shows how CloudLab's userbase has grown along with its capacity. Starting in early 2015, when CloudLab exited its "preview" phase and became open for general use, it has grown steadily. As its capacity has increased, so too have its users: the more capacity, the more active users there are at a time, and the more projects (roughly corresponding to research groups and classes) are supported. Within the general upward trend, specific yearly cycles can be seen. CloudLab has lower usage during the summer, when there are few classes and research activity is slow. CloudLab's peak usage typically comes in the late spring: this is due to the confluence of major paper deadlines (OSDI, SOSP, SOCC, NSDI spring deadline, etc.) and end-of-year coursework.

CloudLab needs to gracefully handle periods of both high and low utilization. While we expected variation in usage over time, the extent to which it drives user behavior was somewhat surprising. Because most of CloudLab's usage is interactive, periods of low utilization are simply times when users are able to start experiments at will, without having to wait for resources to become available. We have therefore focused on improving user experience during periods of heavy utilization: as detailed in Section 3.3, we have built an *optional* reservation system which allows users to schedule resources ahead of time. Another strategy would be to incentivize users to shift their work from periods of high demand to periods of lower demand. In commercial clouds, one way of doing this is through spot pricing [1], which offers economic incentives. Because CloudLab's users do not pay to use it, economic incentives are not available; while various "virtual currency" approaches have been proposed for use in related facilities such as PlanetLab [18], none have seen widespread

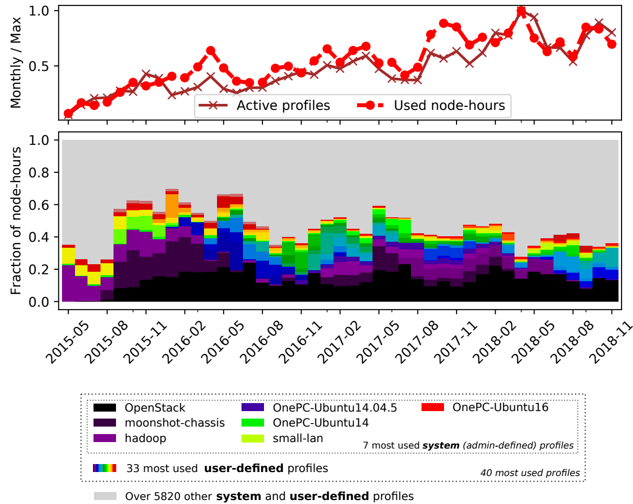


Figure 2: Evolving distribution of profile usage.

success. Demand in CloudLab's core communities appears hard to shift: Figure 1 suggests that CloudLab's users are heavily deadline-driven; it is not feasible to do coursework outside of the school-year, and not many research projects can run their experiments months before conference deadlines. If CloudLab is to "fill in" lightly-loaded periods, it will need to adopt backfill strategies [11, 23] that enable use by work that is easier to shift in time, such as the queueing systems used by high performance and high throughput computing.

Like the number of active users, the number of profiles in use at any time has grown as shown in Figure 2. The bottom half of the figure shows the forty most used profiles, as measured by the cumulative node-hours. We can see that usage patterns vary significantly by month. In some months, such as in early summer 2016, the top forty profiles constitute nearly 70% of the testbed's utilization, while in other months, such as recently, this fraction is less than 40%. The remaining node-hours are allocated by experiments representing over 5,820 other profiles. Users run experiments using up to 572 unique profiles monthly; the median is 233.

The long-tailed distribution we observe indicates that there is a large number of profiles with relatively low and infrequent use, but their combined utilization constitutes most of the user activity. This holds important implications for analysis of usage patterns and for design choices. In Section 2.5, we reflect on the fact that the testbed's major practical value is attributed to facilitating not merely a handful of common use cases, but rather a large variety of experiments in this "long tail." In quantitative analysis, it means we should use medians rather than means as the preferred measure of central tendency due to the highly skewed distributions. The analysis of other percentiles (e.g., 75th or 95th percentiles) provides complementary insights, as we discuss in Section 3.3. Furthermore, this diverse and evolving utilization distribution suggests that we cannot draw reliable conclusions about the impact of testbed's capabilities on usage patterns based solely on comparison of usage statistics from different periods of time. For example, if we compare statistics for 2017 and 2018, it would be difficult to determine the extent to which evolving usage patterns were due to changes in the system or due to the natural evolution of user interests. For the same reason, month-to-month comparisons are also unlikely to provide sufficient evidence for "before and after" analyses for system capabilities.

Fundamentally, periods of time that seem similar by one statistical measure (such as the number of active users) can look very different for other measures (such as the distribution of profile use). We posit that this is likely to be true for many IaaS-type facilities: while multi-tenancy smooths out some measures into predictable shapes (e.g., the top half of Figure 2), others are quite chaotic (e.g., the bottom half of the same figure). Taking these patterns into account when designing facilities can improve their utility and user experience.

2.4 Resource Availability

Availability and diversity of resources play critical roles in the adoption and continued use of a testbed or other IaaS facility. If users' needs frequently cannot be satisfied due to insufficient availability, those users will likely move to other facilities. Users also tend to seek out hardware with cutting-edge features and the highest performance characteristics. New hardware types have been introduced to CloudLab over time in order to satisfy both capacity and the capability requirements. Not only did the new hardware attract new users, but it also reduced contention for older, already deployed resources.

In Figure 3, we show both short- and long-term availability trends for the major CloudLab hardware types. The X-axis represents a fraction of all nodes of a particular type, and the Y-value at each point shows what fraction of the time *at least* that many nodes were available. Lines on these graphs that fall steeply signify the types that are in use most of the time, while higher curves represent more available types. For example, we can compare `d430` and `m400`: the former type is more heavily used across all three graphs.

As we saw in the previous analysis, metrics that look smooth when viewed from a high level show much more variability when we look at the details. This is important because it is often the details that influence users' experience: e.g., for an individual user, availability of the specific node type(s) needed for their experiment is important, rather than the availability of the testbed as whole. To illustrate this, we include the two monthly plots showing the variation that occurs between "slow" and "busy" months. For example, the curves for `pc3000` indicate that in January 2018 users found 80% of these nodes available for use 80% of the time. In contrast, in April of that year, there were *no* times at which 80% of these nodes were available. For several other hardware types (such as `c220g1` and `d710`), resources were even

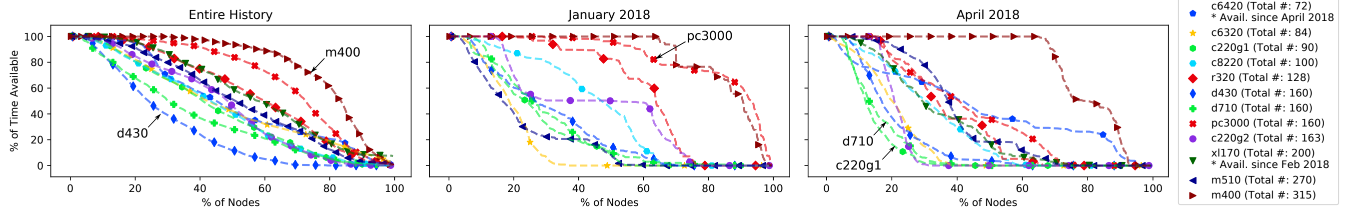


Figure 3: The availability of CloudLab resources. The left plot shows availability over the entire period of time each hardware type was available; the other two plots show availability during low-utilization (center) and high-utilization (right) months.

scarcer, and at no time are more than 30–40% of these nodes free. We also note that d430's low availability is reflected in two ways: it can be seen in these graphs, and we will discuss it in Section 3.3, where we find that users commonly make reservations to schedule access to this in-demand hardware. The curves in these figures highlight that the extent to which system changes and evolving utilization patterns impact individual hardware types varies significantly across the types.

2.5 Research Use of CloudLab

To understand the research conducted on CloudLab, we surveyed 93 papers published in 2017–2018 that used CloudLab for part or all of their experimental evaluations. Table 1 presents a categorization of the papers by the primary area of contribution, using a list of systems and related research areas. These areas are quite broad, and no one area dominates. Given that the most prevalent area, Networking, is an area where typical clouds provide very little transparency [14] and control only as an overlay [2], it is intuitive that research in this area benefits from CloudLab's greater visibility and control. Research in the second area, Security, benefits from the "closed world" of CloudLab, allowing experiments that involve attacks that would be considered hostile in a typical cloud and defenses that need to be implemented within the cloud framework itself.

In this analysis, we found two primary motivations that drove experimenters to CloudLab. The first is low-level access to hardware because of *features* that could not be developed in virtualized environments. Almost every paper in this set used some different aspect of CloudLab, such as the ability to re-configure Ethernet switches, the ability to build an HPC-like environment with root access, the SDN available on most of the CloudLab networks, the ability to monitor power use, the ability to build a complete OpenStack cloud inside the infrastructure, etc. The second motivation is the *performance predictability and isolation* that are difficult to come by in environments that use multi-tenancy on hosts and storage. When the primary metric of interest in a system is its performance, anything that adds variability requires, at a minimum, a far greater number of experiment repetitions to achieve statistical confidence [22, 17]. There is also the

Networking	30%
Security	16%
Storage	11%
Applications	10%
Computing	9%
Virtualization	8%
Databases	7%
Middleware	4%
Energy & Power	2%
Other	15%

Table 1: Research areas in 93 papers that used CloudLab.

question of the extent to which an evaluation is measuring artifacts of the platform vs. the actual system under test: while a more transparent environment does not guarantee that no system artifacts are present, it does give the experimenter more opportunities to observe, understand, and correct for these effects.

The main lesson we take from this analysis is that, as facility operators, we are constantly surprised by the uses to which users put the facility. Had we started from a position of virtualizing everything, then providing lower-level access to specific systems as needed, we think it is unlikely that we would have been able to anticipate all of the use cases found in this survey. Starting from a position of maximizing user control helps to maximize use of the facility.

3 High-Level Effects of Low-Level Decisions

We now move from examining how people use CloudLab to looking at the interactions between design decisions, operational experience, and user behavior. We have found that the *choice of algorithms* deep within the implementation of a system like CloudLab has a profound effect on the ways that users interact with the system, and even what they are able to accomplish. As a result, we have made many changes to the CloudLab facility during its lifetime; in this section, we present individual subsystems that we had to evolve based on facility's usage patterns. The high-level theme of this section is that the choices made at these low levels are not, contrary to what one might expect, simply implementation details, nor are they neutral with respect to the utility of the facility. When building an IaaS facility, designers cannot consider aspects such as resource mapping separately from user goals, require-

ments, and workflows. These aspects of the system must be co-designed, so that users can work *with* these subsystems rather than having to fight *against* them to get work done.

3.1 Resource Mapping

There exist several approaches to the problem of mapping user requests to physical resources. For instance, commercial clouds do not provide control over this mapping within selected instance classes; they manage the placement and consolidation for effective utilization and hide the details from the users. In contrast, Chameleon [21], which is designed as a testbed for repeatable experiments (similar to CloudLab but serving a different research community), has its users do the mapping by asking them to specify IDs of the particular servers they want to use in their requests.

CloudLab takes a unique approach where it recognizes two aspects in this mapping. It is a constraint-satisfaction problem in the sense that the user's request is a specification that must be satisfied; specifically, it resembles the subgraph isomorphism problem [10] in that both the requested and physical topologies are graphs consisting of servers, switches, etc. It is also an optimization problem, because the mapping must be done in a way that maximizes the possibilities for future mappings: it should avoid using scarce resources unless they are specifically requested or there is no available alternative. CloudLab exposes the outcomes of the mapping to the users and allows them to reuse hardware IDs if necessary.

CloudLab's mapping algorithm is derived from the one developed for Emulab [29], and uses simulated annealing to address this NP-hard problem. The advantage of using a powerful, general-purpose algorithm is that it enables the expression of complex constraints and preferences. The disadvantage, however, is that when a mapping *cannot* be found for a request, it can be difficult for users—and even administrators—to understand why. In CloudLab, we have had to evolve this system to improve the intelligibility of the responses that it provides.

The fundamental trade-off exposed here is between a *general* algorithm that makes few assumptions about the facility (and therefore is easily adaptable to new resources) and a more *specialized* algorithm that understands facility semantics and can provide actionable suggestions when a mapping fails. The general algorithm fundamentally lacks semantic information about what the user *may be trying to accomplish* and the *classes of requests that "make sense" on this particular testbed*. A mapping algorithm more tailored to a specific use case could embed such information and make assumptions about user goals.

Our response to this trade-off has been to retain the general algorithm, but to develop a set of heuristics that turn some of the more common failure modes into messages that are easier for users to understand. A major challenge in designing these heuristics is that they must be *conservative*: that is, every mapping that would have succeeded without the heuristic must still succeed. Our experience has been that it is preferable to build such heuristics *around* the mapping algorithm rather than *into* it. Building conservative checks into the *randomized* setting of the mapper itself is extremely difficult and can easily cause unexpected changes in behavior. It is easier—and more informative for the user—to build conservative checks as a *deterministic* wrapper around the mapper. We now describe some of these checks, which we have added over time in response to common error patterns and common questions from users.

In an ideal situation, all mapping errors would be explained to the user by concise, actionable error messages. In theory, the universe of possible mapping errors is so vast that not all have simple explanations. We have found that in practice, however, it is possible to catch most mapping errors with heuristics. We now describe the set of heuristics we have developed over time in response to use patterns and frequent user questions.

Looking in Table 2 at the last year (L.Y.) of mapping errors, approximately 84% of all errors are explained by the top 10 error messages, and of that top 10, only 13.5% are ones that we classify as "unhelpful." If we look at this as a percentage of all experiments, only 1.2% of all attempts to start experiments in the last year have received these four unhelpful mapper messages.

The top two messages (lines 1 and 2 in the table) together account for about half of all mapper errors, and they simply indicate a lack of free nodes (servers or user-controlled switches) at the current time. The first message indicates that there are insufficient nodes free *right now* while the second says that this would occur in the *near future* due to the reservation system described later in this section. There is a third variation on this message (line 7); this is an older version of line 1, which we updated partway through the year to clarify its meaning and provide more specific information. Note that this class of messages are *per-type*, so experiments that request, for example, both servers and user-controlled switches get specific feedback on which is the limitation. The number of available nodes is reported in order to allow the user to decide whether they would prefer to request fewer nodes or to wait until enough nodes become available. When users request specific nodes, in contrast with asking for any nodes of a selected type, they receive explicit messages indicating that those nodes are unavailable (line 10).

Other frequent errors (lines 3, 6, and 8) indicate that there is some node in the request that cannot map to anything available. Our heuristics try to report the specific reason, such as requesting too many physical interfaces, an OS image that is incompatible with the hardware type, or a specific feature (such as a GPU add-on). The distinction between lines 6 and 8 presents an interesting illustration of our use of heuristics: underneath, the mapper uses the same mechanism to handle both of these constraints (support for a particular image is considered a "'feature"). We found that the raw

Error Message	Helpful (actionable)	% of Mapping Errors L.Y.	% of Mapping Errors ALL	% of All Errors L.Y.	% of All Experiments L.Y.
1. Resource reservation violation: X nodes of type HW requested, but only Y available	✓	27.79	14.33	16.07	2.41
2. X nodes of type HW requested, but only Y available nodes of type HW found	✓	21.86	33.01	12.64	1.89
3. No Possible Mapping for X: Too many links of type Y	✓	6.64	6.96	3.84	0.58
4. No Connection	✗	5.22	2.62	3.02	0.45
5. Insufficient Bandwidth	✗	4.88	7.53	2.82	0.42
6. No Possible Mapping for X: OS 'Y' does not run on this hardware type	✓	4.74	3.50	2.74	0.41
7. Not enough nodes because of policy restrictions or existing resource reservations	✓	4.37	2.18	2.53	0.38
8. No Possible Mapping for X: No physical nodes have feature Y	✓	3.54	2.40	2.05	0.31
9. Insufficient Nodes: Unexplained	✗	3.39	2.15	1.96	0.29
10. Fixed physical node X not available.	✓	2.56	3.15	1.48	0.22

Table 2: Distribution of recorded mapping errors. "ALL" denotes the distribution of all errors recorded since October 20, 2015. "L.Y." columns refer to the percentages reported for the last year (starting on August 1, 2017).

message, however, was unhelpful and confusing to users, so we recognize the specific case of image-related mapping failures and transform the message into something that the user can act on: she needs to either pick a different image or a different hardware type.

Lines 4 and 5 are the error messages that are the least helpful to users, and they have a similar cause: the mapper is unable to find a solution that satisfies all links and LANs with the bandwidths specified in the requests. These error messages are produced directly by the simulated annealing portion of the mapper, and it is no coincidence that they are the hardest to explain. They are highly dependent on the details of the topology requested by the user and the switch topology at each CloudLab site. There are many potential actions to take in response to such failures: change the topology, reduce the bandwidth requested, try a different CloudLab site, wait for a different set of physical resources to be free, etc. In essence, the more degrees of freedom the user has with respect to reacting to a failure, the harder it is for the facility to guess which one best addresses the user's actual goals, and the more difficult it is to provide a useful message.

3.2 Interactive Topology Design Feedback

Giving users actionable messages when their profiles don't map is helpful, but it comes fairly late in the process of experiment design. Our experience has been that users can find even the "helpful" mapper errors frustrating, as they come after the user has already invested significant time. A useful analogy is to compile-time errors and syntax checking in IDEs: compiling is complex and slow, and feedback from the editor as the user writes code, while not perfect, leads to a workflow with fewer surprise errors. What we discovered was that we needed the equivalent of realtime syntax checking for network topology design, and our answer to this is CloudLab's *topology constraint system*. The biggest challenge in building it has been to design a system with a simplified model of the mapping process that does not produce a *specific mapping*, but instead checks whether such a solution *should exist*; it must do so quickly enough to run interactively in the browser.

The constraint system is used in two contexts, and has slightly different goals in each. In the first context, it is invoked as part of Jacks: CloudLab's GUI that gives users a "drag and drop" interface for constructing profiles. In this setting, its goal is to assist novice users by disabling UI options that conflict with their existing choices and to warn them when the topology they have drawn is unlikely to be instantiatable. It does not need to admit *every possible* request that can be instantiated on CloudLab (there are more sophisticated interfaces for that), but to provide an assurance that, if a topology passes at this stage, it is virtually guaranteed to succeed in mapping (assuming there are enough resources free). In the second setting, it is used at the final stage of profile instantiation, when the user selects which CloudLab cluster to run their experiment on. Here, it checks the request against each cluster and disables selection of any cluster where the request cannot be instantiated. The goal in this case is the inverse, and we must be more conservative: We want the constraint system to block instantiation if the request will *definitely* fail, but do not want to over-zealously block instantiation that *might* succeed.

The described two-phase experiment design is unique to CloudLab. On the surface, the first phase can be compared to how responsive web interfaces for clouds—e.g., Amazon EC2's dashboard and the OpenStack's Horizon dashboard [27], hide or disable infeasible configurations. At the same time, EC2 goes as far as "attaching" storage characteristics to instance classes (even though networking is actually what is being customized) when listing the storage optimized solutions among the feasible configurations. CloudLab's constraint system makes the design process more explicit by offering interactive control over all components of interconnected experiment environments. In the second phase, requests act analogously to HTCondor's classads [9]. In practice, systems like HTCondor without interactive design capabilities

make working with complex configurations laborious and error-prone.

Generating and checking candidates To check constraints, we generate a set of *candidates* which are tested against a number of *groups*. A candidate is a set of node or link resource properties which we check for mutual compatibility. A group is a whitelist of acceptable combinations relating two or more resource properties. For example, a group might include all allowed combinations of hardware type and disk image. Our constraint system also supports wildcards in both candidates (for unspecified resource properties) and groups (for cases where one resource property is universally allowed). A candidate passes if it matches all groups. Our approach uses a Boolean expression in the *product of sums* form: a set of terms containing conditions that are OR-ed together, with all terms being AND-ed together.

This process is defined in terms of sets and Boolean operations as follows: for a set of candidates $X = \{x_1, x_2, \ldots, x_k\}$, we define an evaluation procedure $f(X)$ that checks all individual candidates. We define $g(x)$ for a given configuration candidate x such that the candidate must match against all groups (A, B, etc.): $g(x) = A(x) \land B(x) \land \ldots$ For each group, the candidate must match at least one condition. As an example, suppose the following table described the conditions allowed for each group:

Group relating `site`, `hardware`, and `type`:	Group relating `hardware` and `image`:
$a_1(x) = \{$`utah`, `m510`, `xen`$\} \subseteq x$	$b_1(x) = \{$`m400`, `ubuntu16-64-ARM`$\} \subseteq x$
$a_2(x) = \{$`utah`, `m400`, `pc`$\} \subseteq x$	$b_2(x) = \{$`m510`, `ubuntu16-64-STD`$\} \subseteq x$
...	...
$a_n(x) = \{$`wisconsin`, `c220g2`, `pc`$\} \subseteq x$	$b_m(x) = \{$`c220g2`, `fbsd110-64-STD`$\} \subseteq x$
$A(x) = a_1(x) \lor a_2(x) \lor \ldots \lor a_n(x)$	$B(x) = b_1(x) \lor b_2(x) \lor \ldots \lor b_m(x)$

In this case, a candidate x=$\{$`utah`, `m400`, `pc`, `ubuntu16-64-ARM`$\}$ evaluates to true, as $a_2(x) \land b_1(x) = 1$.

In the Jacks GUI, the candidates that we generate represent the UI element (node, link, etc.) that the user has selected and the actions they *may* take on it: OS images they may select, other nodes they may connect it to, etc. Each candidate represents a different possible action, and we disable ("gray out") UI elements for candidates that do not pass ($g(x)$ evaluates to false). In the profile instantiation process, the candidates represent *all* nodes as they appear in the request, and the request may only be submitted to clusters for which *all* candidates pass ($f(X)$ evaluates to true).

Checking Constraints Quickly The set of candidates can, in practice, be quite large: in Jacks, it grows with the number of options the user can set on the node (including other nodes to connect to), and in the instantiation process, it grows with the size of the request. We have run containerized experiments with as many as 5,000 nodes. At least one candidate must be evaluated per node in a topology, and if there are LANs, the number of candidates is quadratic in the number of nodes in each LAN. The number of conditions in each group can grow even larger, as it depends in part on the product of the number of hardware types, images, sites, and other node properties. On our current system, every candidate is evaluated against at least 10,000 conditions across all groups. However, the number of groups remains small in all cases (the current number of groups in our testbed is just 7), and in practice, there are several optimizations that allow us to take advantage of the facility environment to make checks fast.

Large requests have many nodes and thus require many candidates to be tested, but many of these candidates will likely be identical. Similarly, when Jacks evaluates which items in a drop-down box are valid, there is no need to re-evaluate combinations that have already been tested on a previous drop-down box instance. Memoizing test results and culling identical candidates yields large speed improvements for our use cases. Even with memoization, every unique candidate has to be checked once, so we have optimized the evaluation of the Boolean expression as well. Naïvely testing each condition in turn using set arithmetic yields a speed that is linear on the number of conditions. Instead, we can uniquely encode conditions as entries in hash tables, and each group can be tested with an (amortized) constant-time lookup. This lookup means that testing a candidate for the first time is linear in the number of *groups* rather than the much larger number of *conditions across all groups*. Together, these optimizations reduce the complexity of the checks from $O(c \cdot g \cdot s)$ (where c is the number of candidates, g is the number of groups, and s is the size of each group) to $O(\text{unique}(c) \cdot g)$.

Impact on User Workflow CloudLab's topology constraint system is built around the idea of using a *quantitative* advantage (fast constraint checking) to provide a *qualitative* improvement in user experience. It has done so by dramatically reducing the number of submitted requests that could not possibly map—even if all resources on the testbed were available. In many situations, builders of IaaS-type facilities face a choice: to ensure that *any* request that a user makes for *any* set of resources configured in *any* way can be instantiated on the facility, or to constrain user requests in some way. While the former is attractive, it can be expensive to guarantee and can result in situations where users *can* request certain combinations but would be better off *not* doing so because these combinations do not perform well together. CloudLab's topology constraint system shows one possible path forward on the latter alternative: constrain users' requests, and give them early, interactive feedback while they design their configurations.

3.3 Reserving Resources

Until recently, resource allocation in CloudLab was done in a First-Come-First-Served (FCFS) manner. While FCFS works well for the interactive "code, compile, debug, gather results" workflow used in the systems research community, it has a number of shortcomings: it favors small experiments

(whatever fits into the available resources at the time the user is active), it can be difficult to plan for deadlines (such as the paper and class deadlines seen in Section 2.3), and it can be problematic for events that must occur at a specific time (such as tutorials and demonstrations). In response to these competing needs, we have developed a *reservation system* to support these use cases while continuing to support the dominant FCFS model.

A reservation is not an experiment scheduled to run at a specific time, but a guarantee of *available resources* at that time. This allows users to run many experiments either in series (e.g., to test different scenarios) or in parallel (e.g., one experiment per student in a class). This *loose experiment-reservation coupling* is one of the key design attributes of our reservation system and the subject of much of the analysis presented in this section.

What we found in designing our reservation system was that it needed to have a fundamentally different design than the resource mapping described in Section 3.1. Resource mapping answers the question, "Given a specific request and a set of available resources, which ones should we use?" The reservation system needs to answer "Given the current schedule of experiments and reservations, would a given action (creating a new experiment, extending an existing one, or creating a new reservation) violate that schedule?" Answering this question must be fast: like the constraint system, we need the reservation system to run at interactive speeds so that we can give users immediate feedback about their ability to create or extend experiments. Our other challenge is to support *late binding* of resources: the reservation system should promise *some* set of resources in the future, but should wait until the time comes to select *specific* ones.

Our approach diverges from the scheduling schemes offered by other facilities. On Chameleon [21], users request specific servers (using server IDs) as mentioned previously; therefore, their requests require only the *early binding*, and the system trades flexibility for simplicity (presumably at the expense of utilization). In contrast, clouds do not offer control over future scheduling decisions. They provide an illusion of infinite resources, and handle all user requests interactively, at the time of submission. In High Performance Computing, solutions are built upon *job queues* where job and user priorities impact scheduling, yet making sure that exact deadlines are met in the future is a constant challenge.

We describe our design using the following terms and operations: A request for reservation r asks for N_r nodes of the specified hardware type h_r to be available within the time window $[s_r, e_r]$. Once `submit`-ed, a request typically requires approval from CloudLab staff, though small requests are auto-approved. In addition to the `approve` operation, staff can `delete` reservations, both pending and active. At any time, users can change their experimentation plans and `delete` their reservations or submit modified requests.

Late Binding Considering that CloudLab's hardware is homogeneous within each hardware type h, the reservation system does not need to decide which *specific* nodes will be counted as N_r nodes of type $h_r \in \{h\}$: any N_r such nodes will satisfy the needs of reservation r with these parameters. This increases efficiency of resource use and helps accommodate FCFS users: it does not require us to force experiments out just because the specific nodes they have allocated happen to be reserved. As long as there are *enough* free nodes for everyone who has requested them, all experiments can continue. Therefore, we spare the reservation system the task of finding exact mappings between reservations and specific nodes and implement reservation operations as node counting tasks. The "binding" occurs later, when the user instantiates their experiment(s) near or within the $[s_r, e_r]$ window. The reservation system simply ensures that the capacity is sufficient.

Checking Reservations Quickly Given the data about active experiments—node counts and their current expiration times—and parameters of approved upcoming reservations, our reservation system constructs a tentative schedule describing how the number of available nodes is expected to change over time. This schedule can be constructed in $O(n \log n)$ time (it must sort upcoming events by time), and takes $O(n)$ time to check. Here, n is the number of events, which is a sum of the number of current experiments (typically hundreds) and the number of future reservations (typically tens). Effectively, this creates a two-phase process, in which the reservation phase involves tasks that are lightweight and fast, while the laborious resource mapping phase runs as part of lengthy resource provisioning process.

This fast checking is enabled by a key design decision: reservations are *per hardware type*—we do not allow reservations for broader categories such as "any server type." While the latter would be attractive, it would also raise the time to check the schedule far above $O(n)$. In our design, we can check the schedule for each type *independently* because the sets of nodes of each type do not overlap. There is only one, binary solution at each point in the schedule: either the sum of nodes in experiments plus the reservations exceeds the total number of nodes of that type, or it does not. If we were to have overlapping sets (e.g., specific and generic types), this would create *dependencies* both between sets and across time. Each point in the schedule would have multiple potential solutions, using different numbers of nodes from each node set. Checking the solution would not only be a matter of checking the solution at each point in time, but ensuring each solution is consistent with the solutions at other time points. The combinatorial complexity that this would entail would prevent us from quickly re-calculating and checking schedules, so we accepted the tradeoff of being more rigid with respect to node types.

Enforcing Reservations The CloudLab reservation system essentially works by "accumulating" free nodes up to the

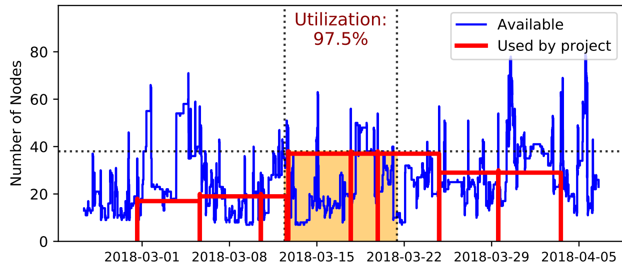


Figure 4: 38 d430 nodes reserved and used for over 9 days. The highlighted box depicts the reserved resources: the number of nodes (up to the horizontal dotted line) reserved for the reservation's time window (between the vertical dotted lines).

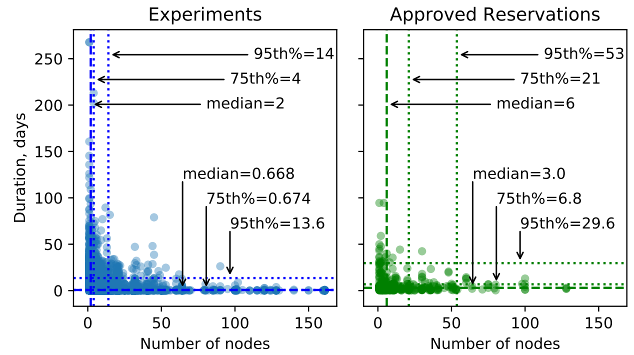


Figure 5: Comparing experiments and reservations. Empirical percentiles are labeled using '%'.

point at which the reservation starts. As the beginning of the reservation approaches, it scrutinizes two types of operations: creation of new experiments, and extending the duration of existing experiments. If either of these operations would overlap with the reservation and would result in there being insufficient free nodes to satisfy it, they are denied. For an illustration, refer to Figure 4, which shows a large reservation *r* requesting over 8200 node-hours worth of d430 nodes. Prior to *r*, d430's availability was insufficient for most of the preceding week (below the horizontal line). As *r*'s start approached, the admission control system began denying overlapping use, and the free nodes rose until the reservation could be satisfied. Almost immediately, the project created a large experiment (the exact size of *r*), and ran two other subsequent experiments. We can see that the final experiment outlasted the reservation: because there were no other reservations directly afterwards, the user was allowed to extend the duration of the experiment. Another interesting behavior visible in this graph is that the project was running smaller experiments before their reservation started; once it did start, they were able to double the size of their experiments.

Parameter Exploration In addition to submit, approve, and delete, CloudLab's reservation system supports a validate operation. validate allows users to explore potential reservations without submitting them, giving them the ability to try different times, hardware types, and reservation sizes to find configurations that fit their needs. If a validation succeeds, the user may submit the reservation. Taking a cue from our mapping and constraint systems, the validation procedure provides users with actionable feedback when the validation fails: messages take the form "Insufficient free nodes at Fri Sep 21 18:00:00 2018 (12 more needed)." This feedback suggests that reducing the number of nodes, shortening the reservation's duration, or moving the reservation further into the future can help the user proceed with submitting a valid reservation.

To understand how users explore different possibilities, we analyzed operations performed on our reservation system between December 6, 2017 and November 30, 2018. Among the 3,500 events in this dataset, there are approximately 1,800 validate and 900 submit operations. These events represent the activity of 200 users working on over 130 unique projects. Nearly 51% of reservation events are validate operations. On average, 2.1 validations preceded each submitted reservation (at least one is required, because users must validate a reservation before they submit it). Further analysis of the operations uncovers infrequent but revealing scenarios. For each submit-ed reservation, we consider validate operations preceding it to be indicative of a user exploring possible candidate reservations. These form a long-tailed distribution: 71% of submit operations were preceded by a single validate, and 14% by two. The remaining 15% of this distribution stretches to a maximum of 32 validate trials. We interpret such cases as empirical evidence for the validation procedure being sufficiently fast to allow users repeatedly *check* and *update* reservation parameters when searching for combinations that satisfy both their needs and the testbed's schedule.

Size and Duration We next compare reservations with experiments, to see whether reservations succeed in enabling larger experiments than are possible with FCFS alone. Using the same time period and reservation data as the previous analysis, we also look at records for 33,300 experiments. Figure 5 illustrates the long-tailed distributions we observe in both. Because these distributions are highly skewed, we characterize and compare them using medians (i.e., 50th percentiles), 75th, and 95th percentiles. The ratios between the pairs of the corresponding percentiles indicate that the reservations are 2.2–10.2 times larger and 3.0–5.3 times longer than experiments. We conclude that reservations do indeed enable larger experiments, though interestingly, the largest experiments were larger than the largest reservations by about 50%. Our analysis of monthly distributions also reveals that the 95th percentile for experiment durations shows significantly less volatility after we introduced the reservation system and stabilizes at its high values, around 300 hours. The same is not true of the node count statistics; the timing of the largest

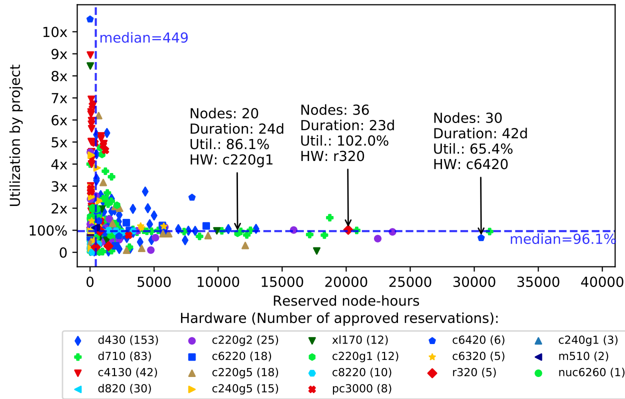


Figure 6: Utilization of reservations.

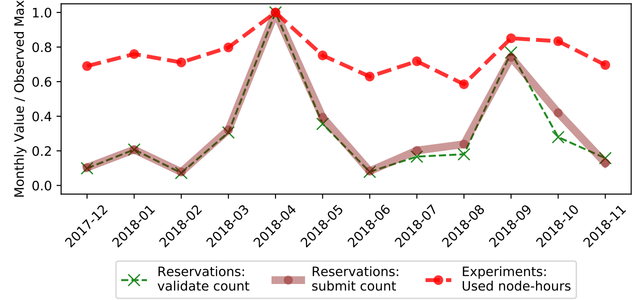


Figure 7: Use of reservations and experiments. For each metric, we divide monthly values by the all-time maximum.

spikes suggests that they are caused by testbed expansions.

Utilization CloudLab does not automatically instantiate experiments for users at the beginning of a reservation, nor does it require the end of a reservation to coincide with the end of an experiment. The current system has no direct penalties for under-using reserved nodes. This leads to an obvious utilization question: *How fully do experimenters use their reservations?* Put another way, this can be framed as a question of trust: *Can we trust users to reserve only what they need and then use what they have reserved?*

Before answering this question, we note several operational nuances that stem from the loose coupling between experiments and reservations in our design. First, we do not stop users from allocating *more* resources than they have reserved: the reservation indicates a minimum guaranteed availability, and if more are available, experimenters are free to use them. Second, if multiple experiments run on h_r hardware within $[s_r, e_r]$, we cannot distinguish experiments that are *meant* to use the studied reservation r from the ones that are run opportunistically, in addition to the planned experiments. Third, reservations are associated with *projects* (groups of users), so the user that creates the reservation may or may not be the one who actually uses it. If users in the same project coordinate their activities, one user submits a reservations on behalf of the group; otherwise, when working independently, one or multiple users submit their reservations and run planned experiments, while others run their unrelated FCFS experiments. Since the studied usage record does not allow us to distill exact user intentions, we estimate aggregate project-specific usage of hardware h_r within $[s_r, e_r]$ and view it as the *upper bound* of the intended r's utilization.

Figure 6 visualizes whole-project resource utilization for nearly 450 approved reservations. The highest point, depicting a utilization of almost 11x the quantity of resources reserved, represents a reservation where a single node was reserved for 33 hours. (The figure omits fifteen small reservations that would stretch the Y axis even further, up to 25x.) That reservation was deleted 3 hours into its time window, and, at the same time, the same or other users from the same project ran multi-node experiments up to 32 node-hours in aggregate. The result is that the project as a whole used far more resources during this time than it had reserved. Labels for several large reservations highlight instances where the utilization is near 100%. We find these and many other large experiments that conform well with the the corresponding reservations. With the median utilization for the shown instances at 96.1%, we conclude that the majority of reservations see high utilization, and we can indeed trust users to reserve what they need and use what they reserve.

In contrast, we found a fairly large number of reservations, 123 (not shown on the graph) with no identified usage. These seem to come from the cases in which the users changed their minds but did not delete their reservations, forgot about the reservations (CloudLab does send reminder emails), or, most interestingly, did run experiments but used wrong types of nodes. This final case seems to stem from misunderstandings about either how the reservation system works (the specific hardware type reserved) or the profile being used (the specific hardware type requested). CloudLab does have measures in place to encourage use of the appropriate node type: when the user has a reservation, the cluster selection box defaults to the relevant cluster, and the mapper applies a preference for nodes of the reserved type. Still, it is clear that this is an area for additional work. With the median size at 96 node-hours, however, these unused reservations add up to less than 12% of all node-hours reserved.

Reservations in Action We conclude our discussion of the reservation system by looking at how its use relates to the use of the testbed as a whole. As shown in Figure 7, rises and falls in use of the testbed (as measured by the number of node-hours used per month) are correlated perfectly with rises and falls in submission of reservations. April 2018, CloudLab's busiest month to date (previously seen in Figures 1 and 3) also saw a large spike in reservations: an astonishing 193 requests were submitted that month, or more than six per day. During that month, there were 140k node-hours of approved reservations, as compared with 724k node-hours used in general, telling us that approximately 19% of all node-hours used

that month were used through reservations. During the preceding January, a lighter month, these numbers were 67k, 552k, and 12%, respectively. Another place where the effects of the reservation system appear is Table 2: if we look at the entire time period, simple resource unavailability is the top reason for mapping failures. If we look at just the last year, however, when the reservation system was more stable, better advertised, and more heavily used, node shortages due to upcoming reservations have become more common than "simple" shortages. The April spike was followed by a similar increase in usage in September 2018.

We postulate that, as the use of the testbed approaches its total capacity, (or, as the free resources approach zero), the notional value of a reservation to a user grows super-linearly. By analogy to queuing theory, as the demand rate approaches the service rate, the expected wait time approaches infinity [20]. Facing the possibility that they may have to wait an unbounded amount of time for the resources they need to become available through the FCFS system, users have far greater incentive to submit reservation requests. This results in the pattern that holds true for the aggregate and also specific hardware types. The demand for specific types of nodes fluctuates over time, and users naturally adjust, using reservations only for the types that are in high demand. Overall, our analysis confirms that the reservation system constitutes a successful "social engineering" project on the part of CloudLab in that the system did change user behavior in the desired way: they use reservations heavily during periods of high demand, but then reservations "fade into the background" when they are not needed, letting the traditional FCFS model dominate.

4 Related Work

There is a body of literature focused on design and analysis of computing testbeds. The work that has shaped the research in this area includes the studies of large-scale experimentation environments such as PlanetLab [8], Grid'5000 [6], Emulab [16], Open Cirrus [5], and PRObE [12]. There are also recent studies that examine the Jetstream [33] "production" cloud for science and engineering research, the Chameleon [21] cloud computing testbed, and the Comet [34] supercomputer, among other facilities. These facility studies describe specific needs of research communities, document major design and implementation efforts, and share the unique lessons learned in the process of deploying and operating each system. Our work complements them by describing different aspects of facility operations and yielding insights into different kinds of design decisions. Studies of relevant commercial installations with similar amounts of detail are scarce.

Another relevant theme relates to using academic and commercial cyberinfrastructures to investigate systems topics and solutions with broad applicability, including the topological issues in testbeds [15], performance and repeatability [26, 22], failure analysis [24], individual subsystems such as disk imaging [19, 4] monitoring infrastructure [38], virtualization [16], and cloud federation [13], among others. Our study complements these by focusing on the way that the *control framework* (the software that manages, assigns, and provisions resources), and the abstractions it offers affect user experience and behavior. The key difference from the related work lies in the unique facility- and user-centered scope of our analysis; none of aforementioned facilities has been studied from this angle. Additionally, this paper describes CloudLab's functionality that extends the control framework used in GENI [25, 32], Emulab [39], and Apt [32].

5 Conclusion

Testbeds for computer science research occupy a unique place in the overall landscape of computing infrastructure. They are often used in an attempt to overcome a basic impasse [3]: as computing technologies become popular, research into their fundamentals becomes simultaneously more valuable and more difficult to do. The existence of production systems such as the Internet and commercial clouds motivates work aimed at improving them, but production deployments offer service at a specific layer of abstraction, making it difficult or impossible to use them for research that seeks to work *under* that layer or to change the abstraction significantly.

The design and operation of testbeds—and other IaaS infrastructures—benefits greatly from analyzing data about how these facilities are used. In this paper, we have presented new analysis of the way that one particular facility, CloudLab, is used in practice. This analysis, and the underlying dataset (which we have made public) have shown that user behavior is highly variable, bursty, and long-tailed. In addition, algorithms that may be thought of as being "deep within" the system have large, visible effects on user experience and on user behavior. Together, these findings point towards design decisions that more carefully take user expectations and behavior into account "end-to-end" throughout the entire facility.

Data and Code

Data and code used for our analyses are available at `https://gitlab.flux.utah.edu/emulab/cloudlab-usage` with the tag `atc19`. This data covers CloudLab's resource availability and events such as experiment instantiations.

Acknowledgments

We thank the anonymous USENIX ATC reviewers and our shepherd, Dilma da Silva, whose comments helped us to greatly improve this work. This material is based upon work supported by the National Science Foundation under Grant Numbers 1419199 and 1743363.

References

[1] Amazon Web Services, Inc. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/.

[2] Amazon Web Services, Inc. Amazon virtual private cloud documentation. https://docs.aws.amazon.com/vpc/index.html.

[3] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4):34–41, April 2005.

[4] K. Atkinson, G. Wong, and R. Ricci. Operational experiences with disk imaging in a multi-tenant datacenter. In *Proceedings of the Eleventh USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2014.

[5] A. I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Ganger, M. A. Kozuch, D. O'Hallaron, M. Kunze, T. T. Kwan, et al. Open cirrus: A global cloud computing testbed. *Computer*, 43(4):35–43, 2010.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[7] M. Brinn, N. Bastin, A. Bavier, M. Berman, J. Chase, and R. Ricci. Trust as the foundation of resource exchange in GENI. In *Proceedings of the 10th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, June 2015.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[9] Computing with HTCondor. HTCondor: Classified Advertisements. https://research.cs.wisc.edu/htcondor/classad/classad.html.

[10] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] D. Duplyakin, D. Johnson, and R. Ricci. The part-time cloud: Enabling balanced elasticity between diverse computing environments. In *Proceedings of the Eighth Workshop on Scientific Cloud Computing (ScienceCloud)*, June 2017.

[12] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX; login*, 38(3), 2013.

[13] N. Grozev and R. Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014.

[14] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. Case study for running HPC applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 395–401, New York, NY, USA, 2010. ACM.

[15] F. Hermenier and R. Ricci. How to build a better testbed: Lessons from a decade of network experiments on Emulab. In *Proceedings of the 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, June 2012.

[16] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the Emulab network testbed. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[17] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[18] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, Aug. 2005.

[19] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning for Clusters. *USENIX ;login:*, 38(1):38–44, Feb. 2013.

[20] L. Kleinrock. *Queueing systems: Theory*. Wiley, New York, 1975.

[21] J. Mambretti, J. Chen, and F. Yeh. Next generation clouds, the Chameleon cloud testbed, and software defined networking (SDN). In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 73–79. IEEE, 2015.

[22] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[23] P. Marshall, K. Keahey, and T. Freeman. Improving utilization of infrastructure clouds. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 205–214. IEEE Computer Society, 2011.

[24] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. 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, June 2014.

[25] R. McGeer, M. Berman, C. Elliott, and R. Ricci, editors. *The GENI Book*. Springer International Publishing, 2016.

[26] L. Nussbaum. Testbeds support for reproducible research. In *Proceedings of the Reproducibility Workshop*, Reproducibility '17, pages 24–26, New York, NY, USA, 2017. ACM.

[27] Rackspace Cloud Computing. Horizon: The OpenStack Dashboard Project. https://docs.openstack.org/horizon/latest/.

[28] Rackspace Cloud Computing. OpenStack: Open source software for creating private and public clouds. https://www.openstack.org/.

[29] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communications Review (CCR)*, 33(2):65–81, Apr. 2003.

[30] R. Ricci, J. Duerig, L. Stoller, G. Wong, S. Chikkulapelly, and W. Seok. Designing a federated testbed as a distributed system. In *Proceedings of the 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, June 2012.

[31] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[32] R. Ricci, G. Wong, L. Stoller, K. Webb, J. Duerig, K. Downie, and M. Hibler. Apt: A platform for repeatable research in computer science. *ACM SIGOPS Operating Systems Review*, 49(1), Jan. 2015.

[33] C. A. Stewart, D. Y. Hancock, M. Vaughn, J. Fischer, T. Cockerill, L. Liming, N. Merchant, T. Miller, J. M. Lowe, D. C. Stanzione, et al. Jetstream: performance, early experiences, and early results. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, page 22. ACM, 2016.

[34] S. M. Strande, H. Cai, T. Cooper, K. Flammer, C. Irving, G. von Laszewski, A. Majumdar, D. Mishin, P. Papadopoulos, W. Pfeiffer, et al. Comet: Tales from the long tail: Two years in and 10,000 users later. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 38. ACM, 2017.

[35] The Mass Open Cloud Team. Mass Open Cloud Web Site. `https://massopen.cloud/`.

[36] The CloudLab Team. CloudLab hardware. `https://www.cloudlab.us/hardware.php`, 2018.

[37] The CloudLab Team. User-controlled switches and layer-1 topologies. `http://docs.cloudlab.us/advanced-topics.html#(part._user-controlled-switches)`, August 2018.

[38] A. Turk, H. Chen, O. Tuncer, H. Li, Q. Li, O. Krieger, and A. K. Coskun. Seeing Into a Public Cloud: Monitoring the Massachusetts Open Cloud. In *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16), Santa Clara, CA*, 2016.

[39] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX, Dec. 2002.

Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure

Bradley C. Kuszmaul Matteo Frigo Justin Mazzola Paluska Alexander (Sasha) Sandler
Oracle Corporation

Abstract

File Storage Service (FSS) is an elastic filesystem provided as a managed NFS service in Oracle Cloud Infrastructure. Using a pipelined Paxos implementation, we implemented a scalable block store that provides linearizable multipage limited-size transactions. On top of the block store, we built a scalable B-tree that provides linearizable multikey limited-size transactions. By using self-validating B-tree nodes and performing all B-tree housekeeping operations as separate transactions, each key in a B-tree transaction requires only one page in the underlying block transaction. The B-tree holds the filesystem metadata. The filesystem provides snapshots by using versioned key-value pairs. The entire system is programmed using a nonblocking lock-free programming style. The presentation servers maintain no persistent local state, with any state kept in the B-tree, making it easy to scale up and failover the presentation servers. We use a non-scalable Paxos-replicated hash table to store configuration information required to bootstrap the system. The system throughput can be predicted by comparing an estimate of the network bandwidth needed for replication to the network bandwidth provided by the hardware. Latency on an unloaded system is about 4 times higher than a Linux NFS server backed by NVMe, reflecting the cost of replication.

1 Introduction

This paper describes Oracle Cloud Infrastructure File Storage Service (FSS), a managed, multi-tenanted NFS service. FSS, which has been in production for over a year, provides customers with an elastic NFSv3 file service [15]. Customers create filesystems which are initially empty, without specifying how much space they need in advance, and write files on demand. The performance of a filesystem grows with the amount of data stored. We promise customers a convex combination of 100 MB/s of bandwidth and 3000 operations per second for every terabyte stored. Customers can mount a filesystem on an arbitrary number of NFS clients. The size of a file or filesystem is essentially unbounded, limited only by the practical concerns that the NFS protocol cannot cope with files bigger than 16 EiB and that we would need to deploy close to a million hosts to store multiple exabytes. FSS provides the ability to take a snapshot of a filesystem using copy-on-write techniques. Creating a filesystem or snapshot is cheap, so that customers can create thousands of filesystems, each with thousands of snapshots. The system is robust against failures since it synchronously replicates data and metadata 5-ways using Paxos [44].

We built FSS from scratch. We implemented a Paxos-replicated block store, called DASD, with a sophisticated multipage transaction scheme. On top of DASD, we built a scalable B-tree with multikey transactions programmed in a lockless nonblocking fashion. Like virtually every B-tree in the world, ours is a B+-tree. We store the contents of files directly in DASD and store file metadata (such as inodes and directories) in the B-tree.

Why not do something simpler? One could imagine setting up a fleet of ZFS appliances. Each appliance would be responsible for some filesystems, and we could use a replicated block device to achieve reliability in the face of hardware failure. Examples of replicated block devices include [2, 4, 25, 54, 61]. We have such a service in our cloud, so why not use it? It's actually more complicated to operate such a system than a system that's designed from the beginning to operate as a cloud service. Here are some of the problems you would need to solve:

- How do you grow such a filesystem if it gets too big to fit on one appliance?
- How do you partition the filesystems onto the appliance? What happens if you put several small filesystems onto one appliance and then one of the filesystems grows so that something must move?

- How do you provide scalable bandwidth? If a customer has a petabyte of data they should get 100 GB/s of bandwidth into the filesystem, but a single appliance may have only a 10 Gbit/s network interface (or perhaps two 25 Gbit/s network interfaces).
- How do you handle failures? If an appliance crashes, then some other appliance must mount the replicated block device, and you must ensure that the original appliance doesn't restart and continue to perform writes on the block device, which would corrupt the filesystem.

This paper describes our implementation. Section 2 provides an architectural overview of FSS. The paper then proceeds to explain the system from the top down. Section 3 describes the lock-free nonblocking programming style we used based on limited-size multipage transactions. Section 4 shows how we organize metadata in the B-tree. Section 5 explains how we implemented a B-tree key-value store that supports multikey transactions. Section 6 explains DASD, our scalable replicated block storage system. Section 7 describes our pipelined Paxos implementation. Section 8 discusses congestion management and transaction-conflict avoidance. Section 9 describes the performance of our system. Sections 10 and 11 conclude with a discussion of related work and a brief history of our system.

2 FSS Architecture

This section explains the overall organization of FSS. We provision many hosts, some of which act as storage hosts, and some as presentation hosts. The storage hosts, which include local NVMe solid-state-drive storage, store all filesystem data and metadata replicated 5-ways,[1] and provide an RPC service using our internal FSS protocol. The presentation hosts speak the standard NFS protocol and translate NFS into the FSS protocol.

A customer's filesystems appear as exported filesystems on one or more IP addresses, called ***mount targets***. A single mount target may export several filesystems, and a filesystem may be exported by several mount targets. A mount target appears as a private IP address in the customer's virtual cloud network (VCN), which is a customizable private network within the cloud. Most clouds provide VCNs in which hosts attached to one VCN cannot even name hosts in another VCN. Each mount target terminates on one of our presentation hosts. A single mount target's performance can be limited by the network interface of the presentation host, and so to get more performance, customers can create many mount targets that export the same filesystem.

[1]Data is erasure coded, reducing the cost to 2.5, see Section 3.

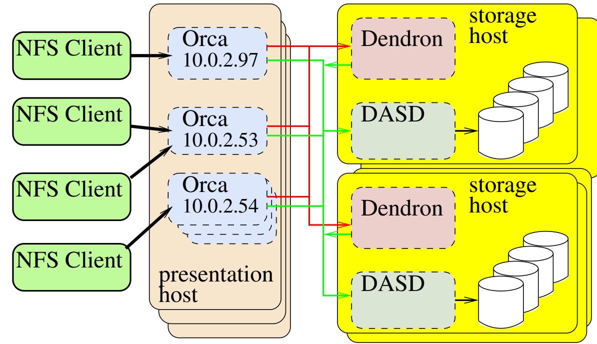


Figure 1: FSS architecture. The NFS clients are on the left, and belong to various customers. Hosts are shown as boxes with solid edges and processes are shown with dashed lines. The presentation hosts are in the middle, each running several Orca processes. The Orca processes are connected to the various customer virtual cloud networks (VCNs) on the left. The IP addresses of each Orca's mount target is shown. The Orca process are also connected to our internal VCN, where they can communicate with the storage hosts. The storage hosts contain NVMe drives and run both the Dendron and DASD processes.

Figure 1 shows how the FSS hosts and processes are organized. The customer sets up NFS clients in their VCN. Our presentation hosts terminate NFS connections from the clients in per-mount-target Orca processes. The Orca processes translate NFS requests into the FSS protocol, and send the FSS to our storage hosts. In the future, the presentation hosts might speak other client protocols, such as SMB [55] or NFSv4 [68].

To ensure isolation between filesystems we depend on a combination of process isolation on our servers, VCN isolation, and encryption. All data stored in the storage hosts or in flight in the FSS protocol is encrypted with a file-specific encryption key that derives from a filesystem master key. The NFSv3 protocol is not encrypted, however, so data arriving at an Orca is potentially vulnerable. To mitigate that vulnerabilty, we rely on VCN isolation while the data is in flight from the NFS client to the presentation host, and use the presentation host's process isolation to protect the data on the presentation host. All data and file names are encrypted as soon as they arrive at an Orca, and each Orca process serves only one mount target.

Each storage host contains NVMe drives and runs two processes, DASD and Dendron. ***DASD***, described in Section 6, provides a scalable block store. ***Dendron*** implements a B-tree (Section 5) in which it maintains the metadata (Section 4) for the filesystem.

We chose to replicate filesystems within a data cen-

ter rather than across data centers within a metropolitan area or across a long distance. There is a tradeoff between latency and failure tolerance. Longer-distance replication means the ability to tolerate bigger disasters, but incurs longer network latencies. We chose local replication so that all of our operations can be synchronously replicated by Paxos without incurring the latency of long-distance replication. It turns out that most of our customers rely on having a functional disaster-recovery plan, and so they're more interested in single-data center file system performance than synchronous replication. In the future, however, we may configure some filesystems to be replicated more widely.

Within a data center, hosts are partitioned into groups called ***fault domains***. We typically employ 9 fault domains. In a small data center, a fault domain might be a single rack. In a large data center, it might be a group of racks. Hosts within a fault domain are likely to fail at the same time (because they share a power supply or network switch). Hosts in different fault domains are more likely to fail independently. We employ 5-way Paxos replicated storage that requires at least 3 out of each group of 5 Paxos instances in order to access the filesystems. We place the Paxos instances into different fault domains. When we need to upgrade our hosts, we can bring down one fault domain at a time without compromising availability. Why 5-way replication? During an upgrade, one replica at a time is down. During that time, we want to be resilient to another host crashing.

We also use the same 5-way-replicated Paxos machinery to run a non-scalable hash table that keeps track of configuration information, such a list of all the presentation hosts, needed for bootstrapping the system.

All state (including NLM locks, leases, and idempotency tokens) needed by the presentation servers is maintained in replicated storage rather than in the memory of the presentation hosts. That means that any Orca can handle any NFS request for the filesystems that it exports. The view of the filesystem presented by different Orcas is consistent.

All memory and disk space is allocated when the host starts. We never run `malloc()` after startup. By construction, the system cannot run out of memory at runtime. It would likely be difficult to retrofit this memory-allocation discipline into old code, but maintaining the discipline was relatively straightforward since the entire codebase is new.

3 Multi-Page Store Conditional

FSS is implemented on top of a distributed B-tree, which is written on top of a a distributed block store with multi-page transactions (see Figure 2). This section describes the programming interface to the distributed block store and how the block store is organized into pages, blocks, and extents.

Customer program
Operating system
NFS
FSS filesystem
B-tree
MPSC
Paxos

Figure 2: Each module is built on the modules below.

The filesystem is a concurrent data structure that must not be corrupted by conflicting operations. There can be many concurrent NFS calls modifying a filesystem: one might be appending to a file, while another might be deleting the file. The filesystem maintains many invariants. One important invariant is that every allocated data block is listed in the metadata for exactly one file. We need to avoid memory leaks (in which an allocated block appears in no file), dangling pointers (in which a file contains a deallocated block), and double allocations (in which a block appears in two different files). There are many other invariants for the filesystem. We also employ a B-tree which has its own invariants. We live under the further constraint that when programming these data structures, we cannot acquire a lock to protect these data structures, since if a process acquired a lock and then crashed it would be tricky to release the lock.

To solve these problems we implemented FSS using a nonblocking programming style similar to that of transactional memory [32]. We use a primitive that we call ***multi-page store-conditional (MPSC)*** for accessing pages in a distributed block store. An MPSC operation is a "mini-transaction" that performs an atomic read-and-update of up to 15 pages. All page reads and writes follow this protocol:

1. Read up to 15 pages, receiving the page data and a ***slot number*** (which is a form of a version tag [38]). A page's slot number changes whenever the page changes. You can read some pages before deciding which page to read next, or you can read pages in parallel. Each read is linearizable [34].
2. Compute a set of new values for those pages.
3. Present the new page values, along with the previously obtained slot numbers, to the MPSC function. To write a page requires needs a slot number from a previous read.
4. The update will either succeed or fail. Success means that all of the pages were modified to the new values and that none of the pages had been otherwise modified since they were read. A successful

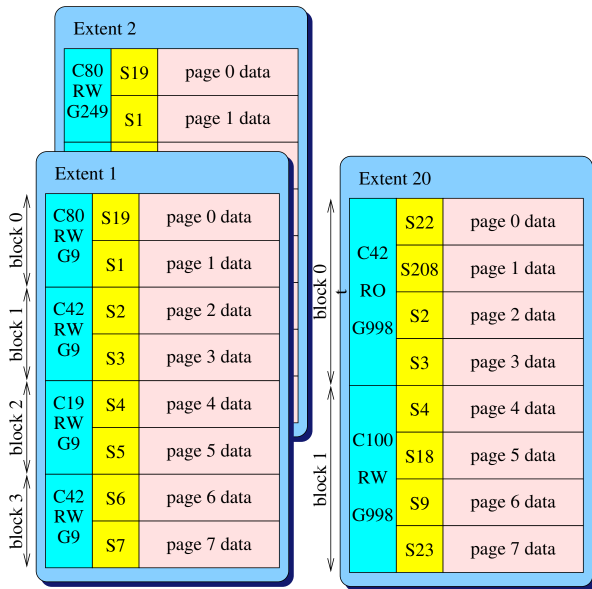


Figure 3: Pages, blocks, and extents. Three extents are shown, each with an array of pages. Each page has a slot. E.g., page 0 of extent 1 has slot 19. Each block has ownership. The first block of extent 1 is owned by customer 80 ("C80"), is read-write ("RW"), and is on its 9th allocation generation ("G9"). Extents 1 and 2 each have 2 pages per block and 4 blocks, whereas extent 20 has 4 pages per block and only 2 blocks.

update linearizes with other reads and MPSC updates. A failure results in no changes.

In addition to reading and writing pages, an MPSC can allocate or free space in the distributed block store.

An MPSC could fail for many reasons. For example, if, between reading a page and attempting an MPSC, some other transaction wrote the page, the MPSC will fail. Even if there is no conflict, an MPSC may fail due to, e.g., packet loss or Paxos leadership changes. Even if a transaction succeeds, the caller may receive an error indication, e.g., if network fails between the update's commit and the caller notification. Our implementation deliberately introduces failures (sometimes called fuzzing [56]) with a small probability rate, so that all of the error-handling code paths are exercised frequently, even in production.

Pages and Blocks. We subdivide the distributed block store into a hierarchy of pages, blocks, and extents, as shown in Figure 3. An MPSC performs an atomic update on a set of ***pages***. A ***block*** includes one or more pages, and is the unit on which we do bookkeeping for allocation. To reduce bookkeeping overheads on small pages, we allocate relatively large blocks. To keep transactions small, we update relatively small pages. An ***extent*** is an array of pages, up to 256 GiB total, and is implemented by a replicated Paxos state machine.

Geometry	Page size	Block size	Extent size	RF	EC
B-tree	8 KiB	1 MiB	16 GiB	5	1
8 KiB	8 KiB	8 KiB	32 GiB	5	5:2
32 KiB	32 KiB	32 KiB	128 GiB	5	5:2
256 KiB	32 KiB	256 KiB	256 GiB	5	5:2
2 MiB	32 KiB	2 MiB	256 GiB	5	5:2

Figure 4: Extent geometries. The B-tree extents contain metadata organized as a B-tree. The other extents contain file contents, and are identified by their block size. For each extent the page size, block size, extent size, replication factor (RF), and erasure-coding (EC) are shown.

For example, one kind of extent contains 256 GiB of disk-resident data, organized in 2 MiB blocks with 32 KiB pages, and is replicated 5 ways using 5:2 erasure coding (an erasure-coding rate of 2/5) [62]. Thus the 256 GiB of disk-resident data consumes a total of 640 GiB of disk distributed across 5 hosts.

An extent's ***geometry*** is defined by its page size, block size, extent size, replication factor, and erasure-coding rate. Once an extent is created, its geometry cannot change. Figure 4 shows the extent geometries that we use for file data and metadata. All of our extents are 5-way replicated within a single data center. The pages in extents used for file contents are erasure coded using a 5:2 erasure coding rate, so that the overhead of storing a page is 2.5 (each replica stores half a page, and there are 5 replicas). The B-tree data is mirrored, which can be thought of as 5:1 erasure coding.

We size our extents so there are hundreds of extents per storage host to ease load balancing. We use parallelism to recover the missing shards when a host crashes permanently—each extent can recover onto a different host.

Block ownership. When operating a storage system as a service, it is a great sin to lose a customer's data. It is an even greater sin to give a customer's data to someone else, however. To avoid the greater sin, blocks have ownership information that is checked on every access.

A block's ownership information includes a version tag, called its ***generation***, as well as 64-bit customer identifier, and a read-only bit. When accessing a block, the generation, customer id, and read-only bit must match exactly. This check is performed atomically with every page access. When a block is allocated or deallocated its generation changes. A ***tagged pointer*** to a page includes the block ownership information, as well as the extent number and page number. A block's pointer is simply the tagged pointer to the block's first page.

The problem that block ownership solves can be illustrated as follows. When data is being written into a new file, we allocate a block and store the block's pointer in the B-tree as a single transaction. To read data from a file, Orca first obtains the block pointer by asking Dendron to read the B-tree. Orca caches that block pointer, so that it can read the data without the overhead of checking the B-tree again on every page access. Meanwhile, another thread could truncate the file, causing the block to be deallocated. The block might then be allocated to a file belonging to a different customer. We want to invalidate Orca's cached pointers in this situation, so we change the block ownership. When Orca tries to use a cached pointer to read a deallocated page, the ownership information has become invalid, and the access fails, which is what we want.

Each of our read operations is linearizable, meaning that they are totally ordered with respect to all MPSC operations and the total ordering is consistent with real time. Although our read operations linearize, if you perform several reads they take place at different times, meaning that the reads may not be mutually consistent. It's easy to trick a transactional-memory-style program into crashing, e.g., due to a failed assertion. For example, if you have two pages in a doubly linked list, you might read one page, and then follow a pointer to the second page, but by the time you read the second page it no longer points back to the first page. Getting this right everywhere is an implementation challenge, leading some [10, 16] to argue that humans should not program transactional memory without a compiler. We have found this problem to be manageable, however, since an inconsistent read cannot lead to a successful MPSC operation, so the data structure isn't corrupted.

4 A Filesystem Schema

This section explains how we represent the filesystem metadata in our B-tree. FSS implements an inode-based write-in-place filesystem using a single B-tree to hold its metadata. What does that mean? "Inode-based" means that each file object has an identifier, called its ***handle***. The handle is used as an index to find the metadata for a file. "Write-in-place" means that updates to data and metadata usually modify an existing block of data. (As we shall see, snapshots introduce copy-on-write behavior.) "Single B-tree to hold the metadata" means there is only one B-tree per data center. Our service provides many filesystems to many customers, and they are all stored together in one B-tree.

The B-tree must support various metadata operations. For example, given an object's handle, we need to find and update the fixed-size part of the object's metadata, which includes the type of the object (e.g., regular, directory, symlink), permissions bits (`rwxrwxrwx`), owner, group, file size, link count, and timestamps. Given a file handle and an offset, we need to find the tagged pointer of the block holding data at that offset, so that reads or write can execute. Given a directory handle and a filename we need to be able perform a directory lookup, yielding a file handle. For a directory, we need to iterate through the directory entries. In NFS this is performed using a 64-bit number called a ***cookie***. Given a directory handle and a cookie we need to find the directory entry with the next largest cookie.

Key-value pairs:

leaderblock:	$0 \to$ next F.
superblock:	$F, 0 \to$ next D, next C, keys.
inode:	$F, 1, D, C, 2, S \to$ stat-data.
name map:	$F, 1, D, C = 0, 3, N, S \to F, D', C', S$.
cookie map:	$F, 1, D, C = 0, 4, c, S \to F, D', C', S, N$.
block map:	$F, 1, D, C, 5, o, S \to$ block ID and size.

Glossary:

F	filesystem number.
D	Directory unique id.
C	File unique id.
S	Snapshot number.
o	Offset in file.
N	Filename in directory.
c	Directory iteration cookie.
F, D', C', S	The handle of a file in a directory.

Figure 5: Filesystem schema showing key $\to$ value pair mappings. The small numbers (e.g., "1") are literal numbers inserted between components of a key to disambiguate key types and force proper B-tree sort ordering. For directories, $C = 0$.

Our strategy is to create B-tree key-value pairs that make those operations efficient. We also want to minimize the number of pages and extents that we access in each transaction. Every B-tree key is prefixed with a filesystem number F, so that all the keys for a given filesystem will be adjacent in the B-tree. Our handles are ordered tuples of integers $\langle F, D, C, S \rangle$, where D a unique number for every directory, C is a unique number for every file ($C = 0$ for directories), and S is a snapshot number. A file's handle depends on the directory in which it was created. The file can be moved to another directory after it is created, but the file's handle will always mention the directory in which the file was originally created.

Figure 5 shows the schema for representing our filesystems. We encode the B-tree keys in a way that disambiguates the different kinds of key value pairs and sorts the key-value pairs in a convenient order. For example, all the pairs for a given filesystem F appear together, with the superblock appearing first because of

the "0" in its key. Within the filesystem, all the non-superblock pairs are sorted by D, the directory number. For a directory, the directory inode sorts first, then come the name map entries for the directory, and then the cookie map, then come all the inodes for the files that were created in that directory. In that set for each file, the file inode sorts first, followed by the block maps for that file. Finally two entries that are the same except for the snapshot number are sorted by snapshot number.

We implement snapshots using copy-on-write at the key-value pair level, rather than doing copy-on-write in the B-tree data structure or at the block level [13, 28, 35, 43, 49, 63, 65, 66, 72]. In the interest of space, we don't show all the details for snapshots, but the basic idea is that each key-value pair is valid for a range of snapshots. When looking up a pair for snapshot S, we find the pair whose key has the largest snapshot number that's no bigger than S.

Our key-value scheme achieves locality in the B-tree. When a file is created it is lexicographically near its parent directory, and the file's block maps and fixed-sized metadata are near each other. (If the file is later moved, it still appears near the original parent directory.) This means that if you create a file in a directory that has only a few files in it, it's likely that the whole transaction to update the directory inode, add directory entries, and create the file inode will all be on the same page, or at least in the same extent, since the B-tree maintains maintains block as well as page locality (see Section 5).

We use multiple block sizes (which are shown in Figure 4) to address the tension between fragmentation and metadata overhead. Small blocks keep fragmentation low for small files. Big blocks reduce the number of block map entries and other bookkeeping overhead for big files. In our scheme the first few blocks of a file are small, and as the file grows the blocks get bigger. For files larger than 16 KiB, the largest block is no bigger than $1/3$ the file size, so that even if the block is nearly empty, we have wasted no more than $1/3$ of our storage. We sometimes skip small-block allocation entirely. For example if the NFS client writes 1 MiB into a newly created file, we can use 256 KiB blocks right away.

5 The B-tree

To hold metadata we built a B-tree [7] on top of MPSC. MPSC provides transactions that can update up to 15 pages, but we want to think about key-value pairs, not pages, and we want B-tree transactions to be able include non-B-tree pages and blocks, e.g., to allocate a data block and store its tagged pointer in a B-tree key-value pair. The B-tree can perform transactions on a total of 15 values, where a value can be a key-value pair, or a non-B-tree page write or block allocation.

Consider the simple problem of executing a B-tree transaction to update a single key-value pair. How many pages must be included in that transaction? The standard B-tree algorithm starts at the root of the tree and follows pointers down to a leaf page where it can access the pair. To update the leaf we need to know that it is the proper page, and the way we know that is by having followed a pointer from the leaf's parent. Between reading the parent and reading the leaf, however, the tree might have been rebalanced, and so we might be reading the wrong leaf. So we we need to include the parent in the transaction. Similarly, we need to include the grandparent and all the ancestors up to the root. A typical B-tree might be 5 or 6 levels deep, and so a single key-value pair update transaction involves 5 or 6 pages, which would limit us to 2 or 3 key-value pairs per transaction. Furthermore, every transaction ends up including the root of the B-tree, creating a scalability bottleneck.

Our solution to this problem is to use ***self-validating pages***, which contain enough information that we can determine if we read the right page by looking at that page in isolation. We arrange every page to "own" a range of keys, for the page to contain only keys in that range, and that every possible key is owned by exactly one page. To implement this self validation, we store in every page a lower bound and upper bound for the set of keys that can appear in the page (sometimes called "fence keys" [26, 47]), and we store the height of the page (leaf pages are height 0). When we read a page to look up a key, we verify that the page we read owns the key and is the right height, in which case we know that if that page is updated in a successful transaction, that we were updating the right page. Thus, we do not need to include the intermediate pages in the MPSC operation and we can perform B-tree transactions on up to 15 keys.

We usually skip accessing the intermediate B-tree nodes altogether by maintaining a cache that maps keys to pages. If the cache steers us to a wrong page, either the page won't self validate or the transaction will fail, in which case we simply invalidate the cache and try again. If a key is missing from the cache, we can perform a separate transaction that walks the tree to populate the cache. It turns out that this cache is very effective, and for virtually all updates we can simply go directly to the proper page to access a key-value pair.

Another problem that could conceivably increase the transaction size is tree rebalancing. In a B-tree, tree nodes must generally be split when they get too full or merged when they get too empty. The usual rule is that whenever one inserts a pair into a node and it doesn't fit, one first splits the node and updates the parent (possibly triggering a parent split that updates the grandparent, and so on). Whenever one deletes a pair, if the node becomes too empty (say less than $1/4$ full), one

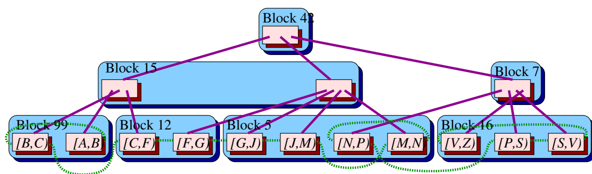


Figure 6: The B-tree comprises block (in blue) and pages (in pink). The pages form a tree. The leaf pages, where the key-value pairs are stored, form a doubly linked list (shown with dashed green lines). Each leaf page is responsible for a range of keys, e.g., $[C,F)$ means the keys from C inclusive to F exclusive. Each block holds a key range of pages for one level. For example, Block 5 has all the leaf pages in the range $[G,P)$.

merges nodes, updating the parent (which can possibly trigger a parent merge that updates the grandparent, and so on). This means that any insertion or deletion can add as many pages to a transaction as the height of the tree. Those rebalancings are infrequent so they don't introduce a scalability bottleneck, but they do make our MPSC operations too big.

Our solution to the rebalancing problem is to rebalance in a separate transaction. When inserting if we encounter a page overflow, we abort the transaction, split the page in a separate transaction, and restart the original transaction. We split the page even if it is apparently nearly empty: as long as there are two keys we can split the page. For merges, we delete keys from the page, and then do a separate transaction afterward to rebalance the tree. It's possible that a page could end up empty, or nearly empty, and that due to some crash or packet loss, we forget to rebalance the tree. That's OK because we fix it up the next time we access the page.

To improve locality we exploit both the page and block structure of MPSC. Figure 6 shows how the B-tree is organized to exploit block locality as well as page locality. Each page is responsible for a key range, and the union of the key ranges in a block is a single key range. When splitting a page, we place the new page into the same block as the old page, and if the block is full, we insert a new block. If the B-tree schema strives to keep keys that will appear in the same transaction lexicographically near each other, locality causes those keys to likely be in the same page, or at least the same block. Our MPSC implementation optimizes for the case where some pages of a transaction are in the same extent. With the schema described in Section 4, this optimization is worth about a 20% performance improvement for an operation such as untarring a large tarball.

The choice of 15 pages per transaction is driven by the B-tree implementation. There is one infrequent operation requiring 15 pages. It involves splitting a page in a full block: a new block is allocated, block headers are updated, and the pages are moved between blocks. Most transactions touch only one or two pages.

6 DASD: Not Your Parent's Disk Drive

This section explains how we implemented MPSC using Paxos state machines (which we discuss further in Section 7). MPSC is implemented by a distributed block store, called ***DASD***[2]. A single extent is implemented by a Paxos state machine, so multipage transactions within an extent is straightforward. To implement transactions that cross extents, we use 2-phase commit.

Given that Paxos has provided us with a collection of replicated state machines, each with an attached disk, each implementing one extent, we implement two-phase commit [29,46,50] on top of Paxos. The standard problem with two-phase commit is that the transaction coordinator can fail and the system gets stuck. Our extents are replicated, so we view the participants in a transaction as being unstoppable.

It would be easy to implement two-phase commit with $3n$ messages. One could send n 'prepare' messages that set up the pages, then n 'decide' messages that switch the state to commited, and then n 'release' messages that release the resources of the transaction. (Each message drives a state transition, which is replicated by Paxos.) The challenge is to implement two-phase commit on n extents with only $2n$ messages and state changes. Every filesystem operation would benefit from the latency being reduced by $1/3$.

To perform an atomic operation with only $2n$ messages, for example on 3 pages, the system progresses through the states shown in Figure 7. The system will end up constructing and tearing down, in the Paxos state machine, a doubly-linked (not circular) list of all the extents in the transaction. Each of these steps is initiated by a message from a client, which triggers a state change in one Paxos state machine (which in turn requires several messages to form a consensus among the Paxos replicas). The client waits for an acknowledgment before sending the next message.

1. Extent A receives a prepare message. A enters the prepared state, indicated by "P(data)", and records its part of the transaction data and its part of the linked list (a null back pointer, and a pointer to B).
2. Extent B receives a prepare message, enters the prepared state, and records its data and pointers to A and C.
3. Extent C receives a prepare-and-decide message, enters the decided state (committing the transac-

[2]Direct-Access Storage Device (DASD) was once IBM's terminology for disk drives [37].

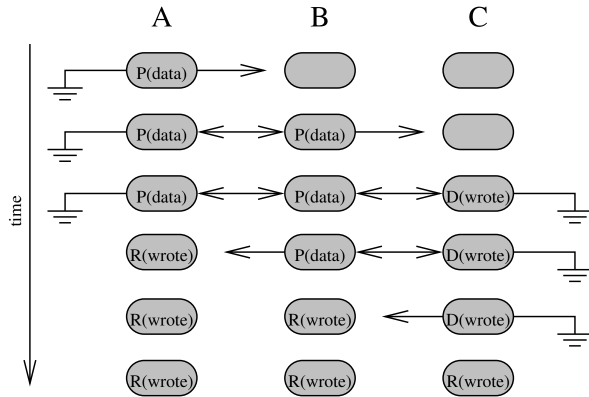


Figure 7: Three extents performing a DASD transaction. Each column is an extent, and each row is a point in time, with time moving downward. A double-arrowed line shows two extents pointing at each other. A single-arrowed line shows one extent pointing at the other, with no pointer back. Ground represents a null pointer.

tion), and records its data and the back pointer to B, indicated by "D(wrote)".

4. Extent A receives a decide-and-release message, notes that the transaction is committed, and releases its resources (such as memory) associated with the transaction, indicated by "R(wrote)". The head of the linked list is now gone.
5. Extent B receives a decide-and-release message, notes the commit, and releases its resources.
6. Extent C receives a release message (it had already decided) and releases its resources.

Thus we implement two-phase commit in exactly $2n$ messages with $2n$ state transitions. Note that the final transition of state C doesn't actually need to be done before replying to the client, and could be piggybacked into the prepare step of the next transaction, potentially reducing the latency to $2n-1$ messages.

The system maintains the invariant that either a prefix or a suffix of the linked list exists, which is useful if the transaction is interrupted. There are two ways that the system can be interrupted, before the commit (in which case the transaction will abort), or after (in which case the cleanup is incomplete). The prefix-suffix property helps in both of these cases. If the transaction gets aborted (at or before step 3) then a prefix exists. If we encounter a prepared state, we can follow the linked list forward until we either find a dangling pointer or a decided state. If we find a dangling pointer, we can delete the prepare record that contained the dangling pointer, preserving a prefix. (At the final point, C, of the linked list, we must extract a promise that C will never decide that the transaction commits. This can be accomplished by introducing a conflict on the read slot for the page.) If we find a decided state then the cleanup was interrupted, so it can proceed back along the linked list until we find the beginning or a dangling pointer, and move the state forward to released.

Our scheme relies on the fact that each state transition occurs one after the other, and hence the critical path of the transition is also $2n$ messages. There are schemes in which one can move the states forward in parallel. For example, one could broadcast "prepare" messages to all the extents, then have one extent decide, and then broadcast decide messages to them all, then release messages, so that the critical path would be only 4 long. This results in $3n$ state transitions (minus one or two, depending on how clever you are.) If you think that big transactions are common, that's valuable, but we have found that most transactions are short so it's better to do the transaction serially.

We optimize the case when there are several pages in a single extent to use fewer messages.

7 Pipelined Paxos

In this section we explain our Paxos implementation, and in particular how we pipeline Paxos operations.

Lamport-Paxos [44, 45] is an algorithm to achieve consensus on a single value. Lamport-Paxos requires two phases, called phase 1 and phase 2 by Lamport.

To achieve consensus on a log (as opposed to one value), one common algorithm is Multi-Paxos [17], which treats the log as an array indexed by slot, running Lamport-Paxos independently on each array element. It turns out that you can run a "vector" phase 1 for infinitely many elements of the array with a single pair of messages, and that you can reuse the outcome of phase 1 for as many phase 2 rounds as you want. In this setup, people tend to call phase-1 "master election" and infer all sorts of wrong conclusions, e.g. that there is only one master at any time and that phase 1 is some kind of "failover".

In Multi-Paxos, if the operation on slot $S+1$ depends on the state after slot S, you must wait for slot S (and all previous slots) to finish phase 2. (We don't say "commit" to avoid confusion with two-phase commit, which is a different protocol.) This Multi-Paxos is not pipelined.

You can pipeline Multi-Paxos with a small modification. You tag each log entry with a unique log sequence number (LSN) and you modify Paxos so that an acceptor accepts slot $S+1$ only if it agrees on the LSN of slot S. Thus, the Paxos phase 2 message is the Lamport phase 2 plus the LSN of the previous slot. By induction, two acceptors that agree on a LSN agree on the entire past history.

Now you can issue phase 2 for $S+1$ depending on S without waiting for S to complete, because the acceptance of $S+1$ retroactively confirms all speculations that you made.

The pipelined Multi-Paxos state, per slot, is the Lamport-Paxos state (a ***ballot*** B and the slot's contents) plus the LSN. You can use whatever you want as LSNs, as long as they are unique, but a convenient way to generate LSNs is to use the pair $\langle E, S \rangle$ where the ***epoch*** E must be unique. As it happens, Lamport phase 1 designates a single winner of ballot B, so you can identify E with the winner of ballot B in phase 1, and be guaranteed that nobody else wins that ballot. In the $E = B$ case, you can reduce the per-slot state to the single-value E, with the dual-role of LSN for pipelining and of ballot for Lamport-Paxos.

Our Paxos algorithm is almost isomorphic to Raft [60]. Essentially Raft is Multi-Paxos plus conditional LSNs plus $E = B$. However, Raft always requires an extra log entry in order to make progress, and cannot be done in bounded space. If you recognize that you are just doing good-old Paxos, then you can make progress by storing a separate ballot B in constant space.

The idea of the acceptance conditional on the previous LSN appeared in viewstamped replication [58] (which didn't discuss pipelining). It is used specifically for pipelining in Zookeeper, except that Zookeeper tries to reinvent Paxos, but incorrectly assumes TCP is an ideal pipe [6]. Conditional acceptance is also used in Raft in the same way as in viewstamped replication, except that Raft lost the distinction between proposer and acceptor, which prevents it from having a speculative proposer state that runs ahead of acceptors.

Recovery. Here we explain how our Paxos system recovers from failures.

The on-disk state of a Paxos acceptor has two main components: the log (of bounded size, a few MB), and a large set of page shards (tens of GB). A shard comprises an erasure-coded fragment of a page and some header information such as a checksum. To write a shard, the Paxos proposer appends the write command to the log of multiple acceptors. When a quorum of acceptors has accepted the command, the write is considered done (or "learned" in Paxos terminology). The proposer then informs acceptors that a command has been learned, and acceptors write the erasure-coded shard to disk.

As long as all acceptors receive all log entries, this process guarantees that all acceptors have an up-to-date and consistent set of shards. However, acceptors may temporarily disappear for long enough that the only way for the acceptor to make progress is to incur a log discontinuity. We now must somehow rewrite all shards modified by the log entries that the acceptor has missed, a process called recovery.

The worst-case for recovery is when we must rewrite the entire set of shards, for example because we are adding a new acceptor that is completely empty. In this ***long-term*** recovery, as part of their on-disk state, acceptors maintain a range of pages that need to be recovered, and they send this ***recovery state*** back to the proposer. The proposer iterates over such pages and overwrites them by issuing a Paxos read followed by a conditional Paxos write, where the condition is on the page still being the same since the read. When receiving a write, the acceptor subtracts the written page range from the to-be-recovered page range, and sends the updated range back to the proposer.

Long-term recovery overwrites the entire extent. For discontinuities of short duration, we use a less expensive mechanism called ***short-term recovery***. In addition to the long-term page range, acceptors maintain a range of log slots that they have lost, they update this range when incurring a discontinuity, and communicate back this slot range to the proposer. The proposer, in the Paxos state machine, maintains a small pseudo-LRU cache of identifiers of pages that were written recently, indexed by slot. If the to-be-recovered slot range is a subset of the slot range covered by the cache, then the proposer issues all the writes in the slot range, in slot order, along with a range R whose meaning is that the present write is the only write that occurred in slot range R. When receiving the write, the acceptor subtracts R from its to-be-recovered slot range and the process continues until the range is empty. If the to-be-recovered slot range overflows the range of the cache, the acceptor falls into long-term recovery.

In practice, almost all normal operations (e.g., software deployments) and unscheduled events (e.g., power loss, network disruption) are resolved by short-term recovery. We need long-term recovery when loading a fresh replica, and (infrequently) when a host goes down for a long time.

Checkpointing and logging. Multi-Paxos is all about attaining consensus on a log, and then we apply that log to a state machine. All memory and disk space in FSS is statically allocated, and so the logs are of a fixed size. The challenge is to checkpoint the state machine so that we can trim old log entries. The simplest strategy is to treat the log as a circular buffer and to periodically write the entire state machine into the log. Although for DASD extents, the state machine is only a few MB, some of our other replicated state machines are much larger. For example we use a 5-way replicated hash table, called Minsk, to store configuration information for bootstrapping the system: given the identity of the five Minsk instances, a Dendron instance can determine the identity of all the other Dendron instances. If the Paxos state machine is large, then checkpointing the state ma-

chine all at once causes a performance glitch.

Here's a simple scheme to deamortize checkpointing. Think of the state machine as an array of bytes, and every operation modifies a byte. Now, every time we log an update to a byte, we also pick another byte from the hash table and log its current value. We cycle through all the bytes of the table. Thus, if the table is K bytes in size, after K update operations we will have logged every byte in the hash table, and so the most recent $2K$ log entries have enough information to reconstruct the current state of the hash table. We don't need to store the state machine anywhere, since the log contains everything we need.

This game can be played at a higher level of abstraction. For example, suppose we think of the hash table as an abstract data structure with a `hash_put` operation that is logged as a logical operation rather than as operations on bytes. In that case every time we log a `hash_put` we also log the current value of one of the hash table entries, and take care to cycle through all the entries. If the hash table contains K key-value pairs, then the entire hash table will be reconstructable using only the most recent $2K$ log entries. This trick works for a binary tree too.

8 Avoiding Conflicts

This section outlines three issues related to transaction conflicts: avoiding too much retry work, avoiding congestion collapse, and reducing conflicts by serializing transactions that are likely to conflict.

In a distributed system one must handle errors in a disciplined fashion. The most common error is when a transaction is aborted because it conflicts with another transaction. Retrying transactions at several different places in the call stack can cause an exponential amount of retrying. Our strategy is that the storage host does not retry transactions that fail. Instead it attempts to complete one transaction, and if it fails the error is returned all the way back to Orca which can decide whether to retry. Orca typically sets a 55 s deadline for each NFS operation, and sets a 1 s deadline for each MPSC. Since the NFS client will retry its operation after 60 s, it's OK for Orca to give up after 55 s.

In order to avoid performance collapse, Orca employs a congestion control system similar to TCP's window-size management algorithm [71]. Some errors, such as transaction conflicts, impute congestion. In some situations the request transaction did not complete because some "housekeeping" operation needed to be run first (such as to rebalance two nodes of the B-tree). Doing the housekeeping uses up the budget for a single transaction, so an error must returned to Orca, but in this case the error does not impute congestion.

When two transactions conflict, one aborts, which is inefficient. We use in-memory locking to serialize transactions that are likely to conflict. For example, when Orca makes an FSS call to access an inode, it sends the request to the storage host that is likely to be the Paxos leader for the extent where that inode is stored. That storage host then acquires an in-memory lock so that two concurrent calls accessing the same inode will run one after another. Orca maintains caches that map key ranges to extent numbers and extent numbers to the leader's IP address. Sometimes one of the caches is wrong, in which case, as a side effect of running the transaction, the storage host will learn the correct cache entries, and inform Orca, which will update its cache. The in-memory lock is used for performance and is not needed for correctness. The technique of serializing transactions that are likely to conflict is well known in the transactional-memory literature [48, 73].

9 Performance

In the introduction we promised customers a convex combination of 100MB/s of bandwidth and 3000 IOPS for every terabyte stored. Those numbers are throughput numbers, and to achieve those numbers the NFS clients may need to perform operations in parallel. This section first explains where those throughput numbers come from, and then discusses FSS latency.

In order to make concrete throughput statements, we posit a simplified model in which the network bandwidth determines performance. The network bottleneck turns out to be on the storage hosts. If VNICs on the NFS clients are the bottleneck, then the customer can add NFS clients. If the presentation host is the bottleneck, then additional mount targets can be provisioned. The performance of the NVMe is fast compared to the several round trips required by Paxos (in contrast to, e.g., Gaios, which needed to optimize for disk latency instead of network latency because disks were so slow [11]).

We define performance in terms of the ratio of bandwidth to storage capacity. There are four components to the performance calculation: raw performance, replication, scheduling, and oversubscription. The ***raw performance*** is the network bandwidth divided by the disk capacity, without accounting for replication, erasure coding, scheduling, or oversubscription.

Replication consumes both bandwidth and storage capacity. Using 5:2 erasure coding, for each page of data, half a page is stored in each of five hosts. This means we can sell only 40% of the raw storage capacity. The network bandwidth calculation is slightly different for writes and reads. For writes, each page must be received by 5 different storage hosts running Paxos. That data is

erasure-coded by each host then written to disk. Thus, for writes, replication reduces the raw network bandwidth by a factor of 5.

For reads we do a little better. To read a page we collect all five erasure-coded copies, each of which is half a page and reconstruct the data using two of the copies. We could probably improve this further by collecting only two of the copies, but for now our algorithm collects all five copies. So for reads, replication reduces the bandwidth by a factor of 5/2.

Scheduling further reduces the bandwidth, but has a negligible effect on storage capacity. Queueing theory tells us that trying to run a system over about 70% utilization will result in unbounded latencies. We don't do quite that well. We find that we can run our system at about 1/3 of peak theoretical performance without affecting latency. This factor of 3 is our ***scheduling overhead***.

Since not all the customers are presenting their peak load at the same time, we can sell the same performance several times, a practice known as oversubscription. In our experience, we can oversubscribe performance by about a factor of 5.

The units of performance simplify from MB/s/TB to s^{-1}, so 100 MB/s/TB is one overwrite per 10 000 s.

For input-outputs per second (IO/s) we convert bandwidth to IOPS by assuming that most IOs are operations on 32 KiB pages, so we provide 3000 IO/s/TB. The cost of other IOs can be expressed in terms of reads: A write costs 2.5 reads, a file creation costs 6 reads, an empty-file deletion costs 8 reads, and a file renaming costs about 10 reads.

This performance model appears to work well on every parallel workload we have seen. To test this model, we measured how much bandwidth a relatively small test fleet can provide. (We aren't allowed to do these sorts of experiments on the big production fleets.) We measured on multiple clients, where each client has its own mount target on its own Orca. This fleet has 41 storage instances each with a 10 Gbit/s VNIC for a total raw performance of 51.25 GB/s. After replication that's 10.25 GB/s of salable bandwidth. Dividing by 3 to account for scheduling overhead is 3.4 GB/s. Those machines provide a total of 200 TB of salable storage, for a ratio of 17 MB/s/TB. According to our model, with 5-fold oversubscription, this fleet should promise customers 85 MB/s/TB.

Figure 8 shows measured bandwidth. The variance was small so we measured only 6 runs at each size. The measured performance is as follows. When writing into an empty file, block allocation consumes some time, and a single client can get about 434 MB/s, whereas 12 clients can get about 2.0 GB/s. When writing into an existing file, avoiding block allocation overhead, the

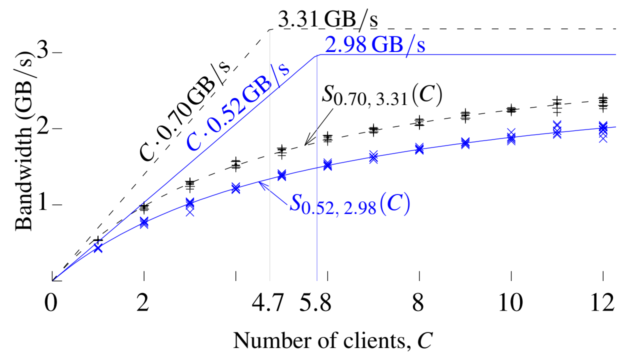


Figure 8: Measured bandwidth. The X-axis is the number of NFS clients. The Y-axis is the cumulative bandwidth achieved. The crosses (in black) show measured performance writing into a preallocated file. The x's (in blue) show measured performance including block allocation. The functions S are the curves fit to a simple-speedup model, with the corresponding linear-speedup shown as lines passing through the origin and the asymptotic speedups shown as horizontal lines. The number of clients at the intercepts are also shown.

performance is about 536 MB/s and 2.4 GB/s for 1 and 12 clients respectively.

We hypothesized that we could model this data as a ***simple-speedup*** curve [9] (a variant of Amdahl's law or of Brent and Graham's Theorem [5,12,27]). In a simple-speedup scenario, as we increase the number of clients, we see a linear speedup which eventually flattens out to give an asymptotic speedup. The curve is parameterized by two numbers l and a. The first value, l, is the ***linear-speedup*** slope which applies when the number of clients C is small where the performance will be $l \cdot C$. The second value, a, is the ***asymptotic speed***, and indicates the performance for large numbers of clients. The simple speedup curve,

$$S_{l,a}(C) = 1/(1/lC + 1/a),$$

is simply half the harmonic mean of the linear-speedup curve lC and the asymptotic speed a.

We fitted of our data to the simple-speedup model and plotted the resulting curves in Figure 8. The asymptotic standard error for the curve fit is less than 1.7%. Visually, the curves fit the data surprisingly well.

We can interpret these curves as follows: When writing to an empty file (which includes block allocation), a few clients can each achieve about 0.52 GB/s, and many clients can achieve a total of 2.98 GB/s. The cutover between "few" and "many" is about 6 clients for this fleet. When writing to a preallocated file, a few clients can each achieve 0.70 GB/s, and many clients can achieve

a total of 3.31 GB/s, which is close to our estimate of 3.4 GB/s. The intercept of the speedup curve lines tells us the ***half-power*** points, where half the peak capacity is consumed: 4.7 clients consume half of the fleet's allocate-and-write capacity, and 5.8 clients consume half of the write-without-allocation capacity.

Low latency is hard to achieve in a replicated distributed system. Latency is the elapsed time from an NFS client's request to response in an otherwise unloaded system. For serial workloads, latency can dominate performance. For example, when running over NFS, the `tar` program creates files one at a time, waiting for an acknowledgment that each file exists on stable storage before creating the next file. After looking at various benchmarks, we concluded that we should simply measure `tar`'s runtime on a well-known tarball such as the Linux 4.19.2 source code distribution, which is 839 MB and contains 65 825 objects. Untarring Linux onto local NVMe device takes about 10 s. The same NVMe served over NFS finishes in about 2.5 minutes. FSS finishes in about 10 minutes. Amazon's EFS, which replicates across a metropolitan region, finishes in about an hour. According to this limited experiment, NFS costs a factor of 15, replication within a datacenter costs another factor of 4, and synchronous replication over metropolitan-scale distances costs another factor of 6. Achieving local-filesystem performance in a replicated distributed fault-tolerant filesystem appears ... difficult.

10 Related Work

MPSC is a variation of load-link/store-conditional [41], and seems less susceptible to the ABA problem (in which the same location is read twice and has the same value for both reads, tricking the user into thinking that no transaction has modified the location in the meanwhile) than compare-and-swap [20,21,33]. Version tagging and the ABA problem appeared in [38, p. A-44].

Sinfonia has many similarities to our system. Sinfonia minitransactions [1] are similar to MPSC. Sinfonia uses uses primary-copy replication [14] and can suffer from the split-brain problem, where both primary and replica become active and lose consistency [19]. To avoid split-brain, Sinfonia remotely turns off power to failed machines, but that's just another protocol which can, e.g., suffer from delayed packets, and doesn't solve the problem. We employ Paxos [44], which is a correct distributed consensus algorithm.

Many filesystems have stored at least some their metadata in B-trees [8, 22, 39, 53, 63, 66, 67] and some have gone further, storing both metadata and data in a B-tree or other key-value store [18, 40, 64, 74, 75]. Our B-tree usage is fairly conventional in this regard, except that we store many filesystems in a single B-tree.

ZFS and HDFS [30,69,70,72] support multiple block sizes in one file. Block suballocation and tail merging filesystems [3,63,66] are special cases of this approach.

Some filesystems avoid using Paxos on every operation. For example, Ceph [42] uses Paxos to run its monitors, but replicates data asynchronously. Ceph's crash consistency can result in replicas that are not consistent with each other. Some systems (e.g., [23, 24, 52]) use other failover schemes that have not been proved correct. Some filesystems store all metadata in memory [24, 31, 36, 42, 57], resulting in fast metadata access until the metadata gets too big to fit in RAM. We go to disk on every operation, resulting in no scaling limits.

11 Conclusion

History: The team started with Frigo and Kuszmaul, and the first code commit was on 2016-07-04. Paxos and DASD were implemented by the end of July 2016, and the B-tree was working by November 2016. Sandler joined and started Orca implementation on 2016-08-03. Mazzola Paluska joined on 2016-09-15 and implemented the filesystem schema in the B-tree. The team grew to about a dozen people in January 2017, and is about two dozen people in spring 2019. Control-plane work started in Spring 2017. Limited availability was launched on 2017-07-01, less than one year after first commit (but without a control plane — all configuration was done manually). General availability started 2018-01-29. As of Spring 2019, FSS hosts over 10,000 filesystems containing several petabytes of paid customer data and is growing at an annualized rate of 8- to 60-fold per year (there's some seasonal variation).

Acknowledgments: In addition to the authors, the team that built and operates FSS has included: the data-plane team of Yonatan (Yoni) Fogel, Michael Frasca, Stephen Fridella, Jan-Willem Maessen, and Chris Provenzano; the control-plane team of Vikram Bisht, Bob Naugle, Michael Chen, Ligia Connolly, Yi Fang, Cheyenne T. Greatorex, Alex Goncharov, David Hwang, Lokesh Jain, Uday Joshi, John McClain, Dan Nussbaum, Ilya Usvyatsky, Mahalakshmi Venkataraman, Viktor Voloboi, Will Walker, Tim Watson, Hualiang Xu, and Zongcheng Yang; the product- and program-management team of Ed Beauvais, Mona Khabazan, and Sandeep Nandkeolyar; solutions architect Vinoth Krishnamurthy; and the engineering-management team of Thomas (Vinod) Johnson, Alan Mullendore, Stephen Lewin-Berlin, and Rajagopal Subramaniyan. Heidi Peabody provided administrative assistance. We further rely on the many hundreds of people who run the rest of Oracle Cloud Infrastructure.

References

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamoanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 27(3), November 2009. doi:10.1145/1629087.1629088.

[2] Alibaba elastic block storage. Viewed 2018-09-26. https://www.alibabacloud.com/help/doc-detail/25383.htm.

[3] Hervey Allen. Introduction to FreeBSD additional topics. In *PacNOG I Workshop*, Nadi, Fiji, 20 June 2005.

[4] Amazon elastic block store. Viewed 2018-09-26. https://aws.amazon.com/ebs/.

[5] G. M. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, 1967.

[6] Apache Software Foundation. Zookeeper internals, December 2009. https://zookeeper.apache.org/doc/r3.1.2/zookeeperInternals.html.

[7] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972. doi:10.1145/1734663.1734671.

[8] Steve Best and Dave Kleikamp. JFS layout. IBM Developerworks, May 2000. http://jfs.sourceforge.net/project/pub/jfslayout.pdf.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[10] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*, pages 15:1–15:6, Berkeley, CA, 30–31 March 2009. https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/boehm/boehm_html/index.html.

[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines at the basis of a high-performance data store. In *Proceedings of the Eighth USENIX Conference on Networked Systems Design and Implementation*, pages 141–154, Boston, MA, USA, 30 March–1 April 2011. http://static.usenix.org/event/nsdi11/tech/full_papers/Bolosky.pdf.

[12] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[13] Gerth Stølting Brodal, Konstantinos Tsakalidis, Spyros Sioutas, and Kostas Tsichlas. Fully persistent b-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12)*, pages 602–614, Kyoto, Japan, 17–19 January 2012. doi:10.1137/1.9781611973099.51.

[14] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems*, chapter 8, pages 199–216. ACM Press/Addison-Wesley, New York, NY, USA, second edition, 1993.

[15] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 protocol specification. IETF RFC 1813, June 1995. https://www.ietf.org/rfc/rfc1813.

[16] Călin Casçaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy. *ACM Queue*, 6(5), September 2008. doi:10.1145/1454456.1454466.

[17] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 398–407, Portland, OR, USA, 12–15 August 2007. doi:10.1145/1281100.1281103.

[18] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E.

Porter, Jun Yuan, and Martin Farach-Colton. File systems fated for senescence? Nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 45–58, 27 February–2 March 2017. https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[19] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned network. *Computing Surveys*, 17(3):341–370, September 1985. doi:10.1145/5505.5508.

[20] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrented deques. In *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*, pages 59–73, 4–6 October 2000.

[21] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, December 2002. Special Issue: Selected papers from PODC'01. doi:10.1017/s00446-002-0079-z.

[22] Matthew Dillon. The hammer filesystem, 21 June 2008. https://www.dragonflybsd.org/hammer/hammer.pdf.

[23] Mark Fasheh. OCFS2: The oracle clustered file system version 2. In *Proceedings of the 2006 Linux Symposium*, pages 289–302, 2006. https://oss.oracle.com/projects/ocfs2/dist/documentation/fasheh.pdf.

[24] GlusterFS. http://www.gluster.org.

[25] Google persistent disk. Viewed 2018-09-26. https://cloud.google.com/persistent-disk/.

[26] Goetz Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems*, 35(3), July 2010. Article No. 16. doi:10.1145/1806907.1806908.

[27] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[28] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[29] Jim N. Gray. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3. Springer-Verlag, 1978.

[30] Add support for variable length block. HDFS Ticket, July 2012. https://issues.apache.org/jira/browse/HDFS-3689.

[31] Hdfs architecture, 2013. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Large_Data_Sets.

[32] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300, San Diego, CA, USA, 16–19 May 1993. doi:10.1145/173682.165164.

[33] Maurice Herlihy. Wait-free synchronizatoin. *ACM Transactions on Programming Languages and Systems (TOPLAS))*, 11(1):124–149, January 1991. doi:10.1145/114005.102808.

[34] 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, July 1990. doi:10.1145/78969.78972.

[35] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, 17–21 January 1994. http://usenix.org/publications/library/proceedings/sf94/full_papers/hitz.a.

[36] Valentin Höbel. LizardFS: Software-defined storage as it should be, 27 April 2016. In German. https://www.golem.de/news/lizardfs-software-defined-storage-wie-es-sein-soll-1604-119518.html.

[37] IBM. *Data File Handbook*, March 1966. C20-1638-1. http://www.bitsavers.org/pdf/ibm/generalInfo/C20-1638-1_Data_File_Handbook_Mar66.pdf.

[38] IBM. *IBM System/370 Extended Architecture—Principles of Operation*, March 1983. Publication number SA22-7085-0. https://archive.org/details/bitsavers_

ibm370prinrinciplesofOperationMar83_40542805.

[39] Apple Inc. Hfs plus volume format. Technical Note TN1150, Apple Developer Connection, 5 March 2004. https://developer.apple.com/library/archive/technotes/tn/tn1150.html.

[40] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4), November 2015. doi:10.1145/2798729.

[41] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, California, November 1987. https://e-reports-ext.llnl.gov/pdf/212157.pdf.

[42] M. Tim Jones. Ceph: A Linux petabyte-scale distributed file system, 4 June 2004. https://www.ibm.com/developerworks/linux/library/l-ceph/index.html.

[43] Sakis Kasampalis. Copy on write based file systems performance analysis and implementation. Master's thesis, Department of Informatics, The Technical University of Denmark, October 2010. http://sakisk.me/files/copy-on-write-based-file-systems.pdf.

[44] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

[45] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4 (Whole Number 121)):51–58, December 2001. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[46] Butler Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, volume 100. Springer Verlag, 1980.

[47] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981. doi:10.1145/319628.319663.

[48] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *The Second ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, USA, 16 August 2007.

[49] A. J. Lewis. LVM howto, 2002. http://tldp.org/HOWTO/LVM-HOWTO/.

[50] Bruce G. Lindsay. Single and multi-site recovery facilities. In I. W. Draffan and F. Poole, editors, *Distributed Data Bases*, chapter 10. Cambridge University Press, 1980. Also available as [51].

[51] Bruce G. Lindsay, Patricia G. Selinger, Cesare A. Galtieri, James N. Gray, Raymond A. Lorie, Thomas G. Price, Franco Putzolu, Irving L. Traiger, and Bradford W. Wade. Notes on distributed databases. Research Report RJ2571, IBM Research Laboratory, San Jose, California, USA, July 1979. http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.

[52] The Lustre file system. lustre.org.

[53] Avantika Mathur, MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 27–30 June 2007.

[54] Microsoft azure blob storage. Viewed 2018-09-26. https://azure.microsoft.com/en-us/services/storage/blobs/.

[55] Microsoft SMB Protocol and CIFS Protocol overview, May 2018. https://docs.microsoft.com/en-us/windows/desktop/FileIO/microsoft-smb-protocol-and-cifs-protocol-overview.

[56] Barton P. Miller, Louis Fredersen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM (CACM)*, 33(12):32–44, December 1990. doi:10.1145/96267.96279.

[57] MooseFS fact sheet, 2018. https://moosefs.com/factsheet/.

[58] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, Toronto, Ontario, Canada, 15–17 August 1988. doi:10.1145/62546.62549.

[59] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA, 19–20 June 2014. https://www.usenix.org/node/184041.

[60] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm (extended version), 20 May 2014. Extended version of [59]. https://raft.github.io/raft.pdf.

[61] Oracle cloud infrastructure block volumes. Viewed 2018-09-26. https://cloud.oracle.com/en_US/storage/block-volume/features.

[62] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathemetics*, 8(2):300–304, June 1960. doi:10.1137/0108018.

[63] Hans T. Reiser. Reiser4, 2006. Archived from the original on 6 July 2006. https://web.archive.org/web/20060706032252/http://www.namesys.com:80/.

[64] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013. https://www.usenix.org/system/files/conference/atc13/atc13-ren.pdf.

[65] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Computational Logic*, 3(4):15:1–15:27, February 2008. doi:10.1145/1326542.1326544.

[66] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3), August 2013. Article No. 9. doi:10.1145/2501620.2501623.

[67] Mark Russinovich. Inside Win2K NTFS, part 1. *ITProToday*, 22 October 2000. https://www.itprotoday.com/management-mobility/inside-win2k-ntfs-part-1.

[68] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network File System (NFS) version 4 protocol. IETF RFC 3530, April 2003. https://www.ietf.org/html/rfc3530.

[69] Chris Siebenmann. ZFS's `recordsize`, holes in files, and partial blocks, 27 September 2017. Viewed 2018-08-30. https://utcc.utoronto.ca/~cks/space/blog/solaris/ZFSFilePartialAndHoleStorage.

[70] Chris Siebenmann. What ZFS gang blocks are and why they exist, 6 January 2018. Viewed 2018-08-30. https://utcc.utoronto.ca/~cks/space/blog/solaris/ZFSGangBlocks.

[71] W. Ricxhard Stevens. TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms. IETF RFC 2001, January 1997. https://www.ietf.org/html/rfc2001.

[72] Sun Microsystems. ZFS on-disk specification—draft, August 2006. http://www.giis.co.in/Zfs_ondiskformat.pdf.

[73] Lingxiang Xiang and Michael L. Scott. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*, pages 234–243, Portland, OR, USA, 13–15 June 2015. doi:10.1145/2755573.2755577.

[74] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *Transactions on Storage—Special Issue on USENIX FAST 2016*, 13(1):3:1–3:21, March 2017. doi:10.1145/3032969.

[75] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 123–138,

Oakland, CA, USA, 12–15 February 2018.
https://www.usenix.org/conference/
fast18/presentation/zhan.

Zanzibar: Google's Consistent, Global Authorization System

Ruoming Pang,[1] Ramón Cáceres,[1] Mike Burrows,[1] Zhifeng Chen,[1] Pratik Dave,[1]
Nathan Germer,[1] Alexander Golynski,[1] Kevin Graney,[1] Nina Kang,[1] Lea Kissner,[2*]
Jeffrey L. Korn,[1] Abhishek Parmar,[3*] Christina D. Richards,[1] Mengzhi Wang[1]
Google, LLC;[1] Humu, Inc.;[2] Carbon, Inc.[3]
{rpang,caceres}@google.com

Abstract

Determining whether online users are authorized to access digital objects is central to preserving privacy. This paper presents the design, implementation, and deployment of Zanzibar, a global system for storing and evaluating access control lists. Zanzibar provides a uniform data model and configuration language for expressing a wide range of access control policies from hundreds of client services at Google, including Calendar, Cloud, Drive, Maps, Photos, and YouTube. Its authorization decisions respect causal ordering of user actions and thus provide external consistency amid changes to access control lists and object contents. Zanzibar scales to trillions of access control lists and millions of authorization requests per second to support services used by billions of people. It has maintained 95th-percentile latency of less than 10 milliseconds and availability of greater than 99.999% over 3 years of production use.

1 Introduction

Many online interactions require authorization checks to confirm that a user has permission to carry out an operation on a digital object. For example, web-based photo storage services typically allow photo owners to share some photos with friends while keeping other photos private. Such a service must check whether a photo has been shared with a user before allowing that user to view the photo. Robust authorization checks are central to preserving online privacy.

This paper presents Zanzibar, a system for storing permissions and performing authorization checks based on the stored permissions. It is used by a wide array of services offered by Google, including Calendar, Cloud, Drive, Maps, Photos, and YouTube. Several of these services manage billions of objects on behalf of more than a billion users.

A unified authorization system offers important advantages over maintaining separate access control mechanisms for individual applications. First, it helps establish consistent semantics and user experience across applications. Second, it makes it easier for applications to interoperate, for example, to coordinate access control when an object from one application embeds an object from another application. Third, useful common infrastructure can be built on top of a unified access control system, in particular, a search index that respects access control and works across applications. Finally, as we show below, authorization poses unique challenges involving data consistency and scalability. It saves engineering resources to tackle them once across applications.

We have the following goals for the Zanzibar system:

- *Correctness*: It must ensure consistency of access control decisions to respect user intentions.
- *Flexibility*: It must support a rich set of access control policies as required by both consumer and enterprise applications.
- *Low latency*: It must respond quickly because authorization checks are often in the critical path of user interactions. Low latency at the tail is particularly important for serving search results, which often require tens to hundreds of checks.
- *High availability*: It must reliably respond to requests because, in the absence of explicit authorizations, client services would be forced to deny their users access.
- *Large scale*: It needs to protect billions of objects shared by billions of users. It must be deployed around the globe to be near its clients and their end users.

Zanzibar achieves these goals through a combination of notable features. To provide flexibility, Zanzibar pairs a simple data model with a powerful configuration language. The language allows clients to define arbitrary relations between users and objects, such as `owner`, `editor`, `commenter`, and `viewer`. It includes set-algebraic operators such as intersection and union for specifying potentially complex access control policies in terms of those user-object relations. For example, an application can specify that users granted editing rights on a document are also allowed to comment on the

*Work done while at Google.

document, but not all commenters are given editing rights.

At runtime, Zanzibar allows clients to create, modify, and evaluate access control lists (ACLs) through a remote procedure call (RPC) interface. A simple ACL takes the form of "user U has relation R to object O". More complex ACLs take the form of "set of users S has relation R to object O", where S is itself specified in terms of another object-relation pair. ACLs can thus refer to other ACLs, for example to specify that the set of users who can comment on a video consists of the users who have been granted viewing rights on that specific video along with those with viewing permissions on the video channel.

Group memberships are an important class of ACL where the object is a group and the relation is semantically equivalent to `member`. Groups can contain other groups, which illustrates one of the challenges facing Zanzibar, namely that evaluating whether a user belongs to a group can entail following a long chain of nested group memberships.

Authorization checks take the form of "does user U have relation R to object O?" and are evaluated by a collection of distributed servers. When a check request arrives to Zanzibar, the work to evaluate the check may fan out to multiple servers, for example when a group contains both individual members and other groups. Each of those servers may in turn contact other servers, for example to recursively traverse a hierarchy of group memberships.

Zanzibar operates at a global scale along multiple dimensions. It stores more than two trillion ACLs and performs millions of authorization checks per second. The ACL data does not lend itself to geographic partitioning because authorization checks for any object can come from anywhere in the world. Therefore, Zanzibar replicates all ACL data in tens of geographically distributed data centers and distributes load across thousands of servers around the world.

Zanzibar supports global consistency of access control decisions through two interrelated features. One, it respects the order in which ACL changes are committed to the underlying data store. Two, it can ensure that authorization checks are based on ACL data no older than a client-specified change. Thus, for example, a client can remove a user from a group and be assured that subsequent membership checks reflect that removal. Zanzibar provides these ordering properties by storing ACLs in a globally distributed database system with external consistency guarantees [15, 18].

Zanzibar employs an array of techniques to achieve low latency and high availability in this globally distributed environment. Its consistency protocol allows the vast majority of requests to be served with locally replicated data, without requiring cross-region round trips. Zanzibar stores its data in normalized forms for consistency. It handles hot spots on normalized data by caching final and intermediate results, and by deduplicating simultaneous requests. It also applies techniques such as hedging requests and optimizing computations on deeply nested sets with limited denormalization. Zanzibar responds to more than 95% of authorization checks within 10 milliseconds and has maintained more than 99.999% availability for the last 3 years.

The main contributions of this paper lie in conveying the engineering challenges in building and deploying a consistent, world-scale authorization system. While most elements of Zanzibar's design have their roots in previous research, this paper provides a record of the features and techniques Zanzibar brings together to satisfy its stringent requirements for correctness, flexibility, latency, availability, and scalability. The paper also highlights lessons learned from operating Zanzibar in service of a diverse set of demanding clients.

2 Model, Language, and API

This section describes Zanzibar's data model, configuration language, and application programming interface (API).

2.1 Relation Tuples

In Zanzibar, ACLs are collections of object-user or object-object relations represented as *relation tuples*. Groups are simply ACLs with membership semantics. Relation tuples have efficient binary encodings, but in this paper we represent them using a convenient text notation:

$\langle tuple \rangle$ ::= $\langle object \rangle$'`#`'$\langle relation \rangle$'`@`'$\langle user \rangle$

$\langle object \rangle$ ::= $\langle namespace \rangle$'`:`'$\langle object_id \rangle$

$\langle user \rangle$ ::= $\langle user_id \rangle$ | $\langle userset \rangle$

$\langle userset \rangle$ ::= $\langle object \rangle$'`#`'$\langle relation \rangle$

where $\langle namespace \rangle$ and $\langle relation \rangle$ are predefined in client configurations (§2.3), $\langle object_id \rangle$ is a string, and $\langle user_id \rangle$ is an integer. The primary keys required to identify a relation tuple are $\langle namespace \rangle$, $\langle object_id \rangle$, $\langle relation \rangle$, and $\langle user \rangle$. One feature worth noting is that a $\langle userset \rangle$ allows ACLs to refer to groups and thus supports representing nested group membership.

Table 1 shows some example tuples and corresponding semantics. While some relations (e.g. `viewer`) define access control directly, others (e.g. `parent`, pointing to a folder) only define abstract relations between objects. These abstract relations may indirectly affect access control given userset rewrite rules specified in namespace configs (§2.3.1).

Defining our data model around tuples, instead of per-object ACLs, allows us to unify the concepts of ACLs and groups and to support efficient reads and incremental updates, as we will see in §2.4.

2.2 Consistency Model

ACL checks must respect the order in which users modify ACLs and object contents to avoid unexpected sharing behaviors. Specifically, our clients care about preventing the

Example Tuple in Text Notation	Semantics
`doc:readme#owner@10`	User 10 is an owner of `doc:readme`
`group:eng#member@11`	User 11 is a member of `group:eng`
`doc:readme#viewer@group:eng#member`	Members of `group:eng` are viewers of `doc:readme`
`doc:readme#parent@folder:A#...`	`doc:readme` is in `folder:A`

Table 1: Example relation tuples. "#..." represents a relation that does not affect the semantics of the tuple.

"new enemy" problem, which can arise when we fail to respect the ordering between ACL updates or when we apply old ACLs to new content. Consider these two examples:

Example A: Neglecting ACL update order

1. Alice removes Bob from the ACL of a folder;
2. Alice then asks Charlie to move new documents to the folder, where document ACLs inherit from folder ACLs;
3. Bob should not be able to see the new documents, but may do so if the ACL check neglects the ordering between the two ACL changes.

Example B: Misapplying old ACL to new content

1. Alice removes Bob from the ACL of a document;
2. Alice then asks Charlie to add new contents to the document;
3. Bob should not be able to see the new contents, but may do so if the ACL check is evaluated with a stale ACL from before Bob's removal.

Preventing the "new enemy" problem requires Zanzibar to understand and respect the causal ordering between ACL or content updates, including updates on different ACLs or objects and those coordinated via channels invisible to Zanzibar. Hence Zanzibar must provide two key consistency properties: *external consistency* [18] and *snapshot reads with bounded staleness*.

External consistency allows Zanzibar to assign a timestamp to each ACL or content update, such that two causally related updates $x \prec y$ will be assigned timestamps that reflect the causal order: $T_x < T_y$. With causally meaningful timestamps, a snapshot read of the ACL database at timestamp T, which observes all updates with timestamps $\leq T$, will respect ordering between ACL updates. That is, if the read observes an update x, it will observe all updates that happen causally before x.

Furthermore, to avoid applying old ACLs to new contents, the ACL check evaluation snapshot must not be staler than the causal timestamp assigned to the content update. Given a content update at timestamp T_c, a snapshot read at timestamp $\geq T_c$ ensures that all ACL updates that happen causally before the content update will be observed by the ACL check.

To provide external consistency and snapshot reads with bounded staleness, we store ACLs in the Spanner global database system [15]. Spanner's TrueTime mechanism assigns each ACL write a microsecond-resolution timestamp, such that the timestamps of writes reflect the causal ordering between writes, and thereby provide external consistency. We evaluate each ACL check at a single snapshot timestamp across multiple database reads, so that all writes with timestamps up to the check snapshot, and only those writes, are visible to the ACL check.

To avoid evaluating checks for new contents using stale ACLs, one could try to always evaluate at the latest snapshot such that the check result reflects all ACL writes up to the check call. However, such evaluation would require global data synchronization with high-latency round trips and limited availability. Instead, we design the following protocol to allow most checks to be evaluated on already replicated data with cooperation from Zanzibar clients:

1. A Zanzibar client requests an opaque consistency token called a *zookie* for each content version, via a *content-change* ACL check (§2.4.4) when the content modification is about to be saved. Zanzibar encodes a current global timestamp in the zookie and ensures that all prior ACL writes have lower timestamps. The client stores the zookie with the content change in an atomic write to the client storage. Note that the content-change check does *not* need to be evaluated in the same transaction as the application content modification, but only has to be triggered when the user modifies the contents.
2. The client sends this zookie in subsequent ACL check requests to ensure that the check snapshot is at least as fresh as the timestamp for the content version.

External consistency and snapshot reads with staleness bounded by zookie prevent the "new enemy" problem. In Example A, ACL updates $A1$ and $A2$ will be assigned timestamps $T_{A1} < T_{A2}$, respectively. Bob will not be able to see the new documents added by Charlie: if a check is evaluated at $T < T_{A2}$, the document ACLs will not include the folder ACL; if a check is evaluated at $T \geq T_{A2} > T_{A1}$, the check will observe update $A1$, which removed Bob from the

folder ACL. In Example B, Bob will not see the new contents added to the document. For Bob to see the new contents, the check must be evaluated with a zookie $\geq T_{B2}$, the timestamp assigned to the content update. Because $T_{B2} > T_{B1}$, such a check will also observe the ACL update $B1$, which removed Bob from the ACL.

The zookie protocol is a key feature of Zanzibar's consistency model. It ensures that Zanzibar respects causal ordering between ACL and content updates, but otherwise grants Zanzibar freedom to choose evaluation timestamps so as to meet its latency and availability goals. The freedom arises from the protocol's at-least-as-fresh semantics, which allow Zanzibar to choose any timestamp fresher than the one encoded in a zookie. Such freedom in turn allows Zanzibar to serve most checks at a default staleness with already replicated data (§3.2.1) and to quantize evaluation timestamps to avoid hot spots (§3.2.5).

2.3 Namespace Configuration

Before clients can store relation tuples in Zanzibar, they must configure their namespaces. A namespace configuration specifies its relations as well as its storage parameters. Each relation has a name, which is a client-defined string such as `viewer` or `editor`, and a relation config. Storage parameters include sharding settings and an encoding for object IDs that helps Zanzibar optimize storage of integer, string, and other object ID formats.

2.3.1 Relation Configs and Userset Rewrites

While relation tuples reflect relationships between objects and users, they do not completely define the effective ACLs. For example, some clients specify that users with `editor` permissions on each object should have `viewer` permission on the same object. While such relationships between relations can be represented by a relation tuple per object, storing a tuple for each object in a namespace would be wasteful and make it hard to make modifications across all objects. Instead, we let clients define object-agnostic relationships via *userset rewrite rules* in relation configs. Figure 1 demonstrates a simple namespace configuration with concentric relations, where `viewer` contains `editor`, and `editor` contains `owner`.

Userset rewrite rules are defined per relation in a namespace. Each rule specifies a function that takes an object ID as input and outputs a userset expression tree. Each leaf node of the tree can be any of the following:

- `_this`: Returns all users from stored relation tuples for the ⟨*object#relation*⟩ pair, including indirect ACLs referenced by usersets from the tuples. This is the default behavior when no rewrite rule is specified.
- `computed_userset`: Computes, for the input object, a new userset. For example, this allows the userset expression for a `viewer` relation to refer to the `editor` userset on the same object, thus offering an ACL inheritance capability between relations.
- `tuple_to_userset`: Computes a tupleset (§2.4.1) from the input object, fetches relation tuples matching the tupleset, and computes a userset from every fetched relation tuple. This flexible primitive allows our clients to express complex policies such as "look up the parent folder of the document and inherit its viewers".

```
name: "doc"

relation { name: "owner" }

relation {
 name: "editor"
 userset_rewrite {
  union {
   child { _this {} }
   child { computed_userset { relation: "owner" } }
} } }

relation {
 name: "viewer"
 userset_rewrite {
  union {
   child { _this {} }
   child { computed_userset { relation: "editor" } }
   child { tuple_to_userset {
     tupleset { relation: "parent" }
     computed_userset {
       object: $TUPLE_USERSET_OBJECT  # parent folder
       relation: "viewer"
   } } }
} } }
```

Figure 1: Simple namespace configuration with concentric relations on documents. All owners are editors, and all editors are viewers. Further, viewers of the parent folder are also viewers of the document.

A userset expression can also be composed of multiple sub-expressions, combined by operations such as union, intersection, and exclusion.

2.4 API

In addition to supporting ACL checks, Zanzibar also provides APIs for clients to read and write relation tuples, watch tuple updates, and inspect the effective ACLs.

A concept used throughout these API methods is that of a *zookie*. A zookie is an opaque byte sequence encoding a globally meaningful timestamp that reflects an ACL write, a client content version, or a read snapshot. Zookies in ACL read and check requests specify staleness bounds for snapshot reads, thus providing one of Zanzibar's core consistency properties. We choose to use an opaque cookie instead of the actual timestamp to discourage our clients from choosing arbitrary timestamps and to allow future extensions.

2.4.1 Read

Our clients read relation tuples to display ACLs or group membership to users, or to prepare for a subsequent write. A read request specifies one or multiple *tuplesets* and an optional zookie.

Each *tupleset* specifies keys of a set of relation tuples. The set can include a single tuple key, or all tuples with a given object ID or userset in a namespace, optionally constrained by a relation name. With the tuplesets, clients can look up a specific membership entry, read all entries in an ACL or group, or look up all groups with a given user as a direct member. All tuplesets in a read request are processed at a single snapshot.

With the zookie, clients can request a read snapshot no earlier than a previous write if the zookie from the write response is given in the read request, or at the same snapshot as a previous read if the zookie from the earlier read response is given in the subsequent request. If the request doesn't contain a zookie, Zanzibar will choose a reasonably recent snapshot, possibly offering a lower-latency response than if a zookie were provided.

Read results only depend on contents of relation tuples and do not reflect userset rewrite rules. For example, even if the `viewer` userset always includes the `owner` userset, reading tuples with the `viewer` relation will not return tuples with the `owner` relation. Clients that need to understand the effective userset can use the Expand API (§2.4.5).

2.4.2 Write

Clients may modify a single relation tuple to add or remove an ACL. They may also modify all tuples related to an object via a read-modify-write process with optimistic concurrency control [21] that uses a read RPC followed by a write RPC:

1. Read all relation tuples of an object, including a per-object "lock" tuple.
2. Generate the tuples to write or delete. Send the writes, along with a touch on the lock tuple, to Zanzibar, with the condition that the writes will be committed only if the lock tuple has not been modified since the read.
3. If the write condition is not met, go back to step 1.

The lock tuple is just a regular relation tuple used by clients to detect write races.

2.4.3 Watch

Some clients maintain secondary indices of relation tuples in Zanzibar. They can do so with our Watch API. A watch request specifies one or more namespaces and a zookie representing the time to start watching. A watch response contains all tuple modification events in ascending timestamp order, from the requested start timestamp to a timestamp encoded in a *heartbeat zookie* included in the watch response. The client can use the heartbeat zookie to resume watching where the previous watch response left off.

2.4.4 Check

A check request specifies a userset, represented by $\langle object\#relation \rangle$, a putative user, often represented by an authentication token, and a zookie corresponding to the desired object version. Like reads, a check is always evaluated at a consistent snapshot no earlier than the given zookie.

To authorize application content modifications, our clients send a special type of check request, a *content-change* check. A content-change check request does not carry a zookie and is evaluated at the latest snapshot. If a content change is authorized, the check response includes a zookie for clients to store along with object contents and use for subsequent checks of the content version. The zookie encodes the evaluation snapshot and captures any possible causality from ACL changes to content changes, because the zookie's timestamp will be greater than that of the ACL updates that protect the new content (§2.2).

2.4.5 Expand

The Expand API returns the effective userset given an $\langle object\#relation \rangle$ pair and an optional zookie. Unlike the Read API, Expand follows indirect references expressed through userset rewrite rules. The result is represented by a *userset tree* whose leaf nodes are user IDs or usersets pointing to other $\langle object\#relation \rangle$ pairs, and intermediate nodes represent union, intersection, or exclusion operators. Expand is crucial for our clients to reason about the complete set of users and groups that have access to their objects, which allows them to build efficient search indices for access-controlled content.

3 Architecture and Implementation

Figure 2 shows the architecture of the Zanzibar system. `aclservers` are the main server type. They are organized in clusters and respond to Check, Read, Expand, and Write requests. Requests arrive at any server in a cluster and that server fans out the work to other servers in the cluster as necessary. Those servers may in turn contact other servers to compute intermediate results. The initial server gathers the final result and returns it to the client.

Zanzibar stores ACLs and their metadata in Spanner databases. There is one database to store relation tuples for each client namespace, one database to hold all namespace configurations, and one changelog database shared across all namespaces. `aclservers` read and write those databases in the course of responding to client requests.

`watchservers` are a specialized server type that respond to Watch requests. They tail the changelog and serve a stream of namespace changes to clients in near real time.

Zanzibar periodically runs a data processing pipeline to perform a variety of offline functions across all Zanzibar data in Spanner. One such function is to produce dumps of the relation tuples in each namespace at a known snapshot times-

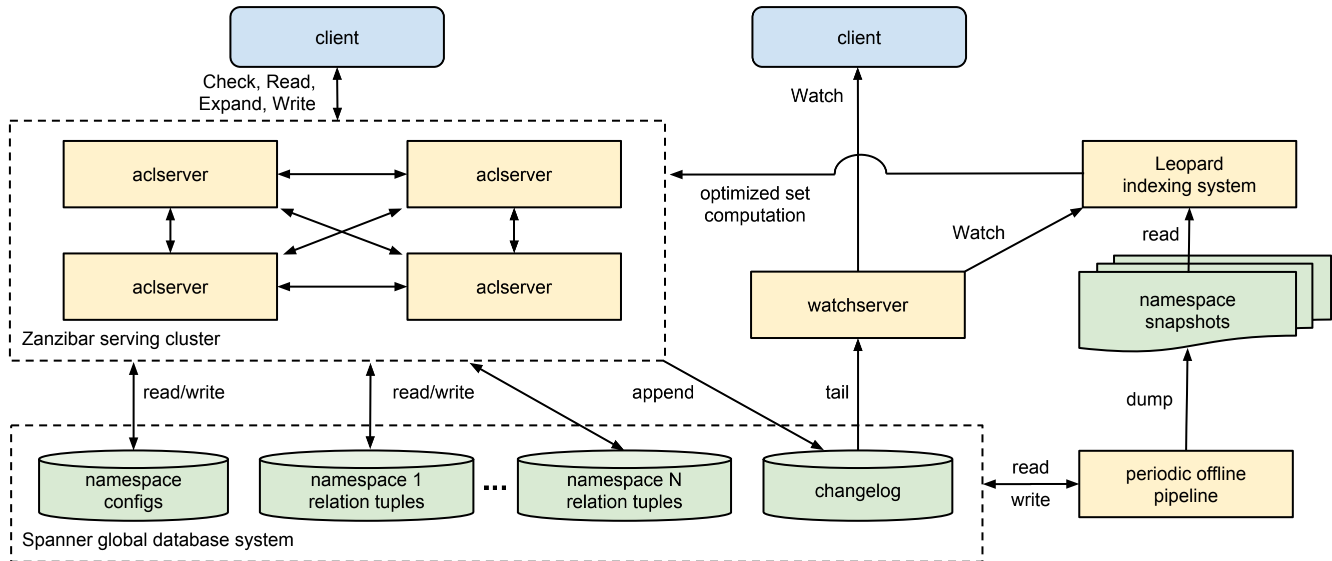


Figure 2: Zanzibar architecture. Arrows indicate the direction of data flow.

tamp. Another is to garbage-collect tuple versions older than a threshold configured per namespace.

Leopard is an indexing system used to optimize operations on large and deeply nested sets. It reads periodic snapshots of ACL data and watches for changes between snapshots. It performs transformations on that data, such as denormalization, and responds to requests from `aclservers`.

The rest of this section presents the implementation of these architectural elements in more detail.

3.1 Storage

3.1.1 Relation Tuple Storage

We store relation tuples of each namespace in a separate database, where each row is identified by primary key *(shard ID, object ID, relation, user, commit timestamp)*. Multiple tuple versions are stored on different rows, so that we can evaluate checks and reads at any timestamp within the garbage collection window. The ordering of primary keys allows us to look up all relation tuples for a given object ID or *(object ID, relation)* pair.

Our clients configure sharding of a namespace according to its data pattern. Usually the shard ID is determined solely by the object ID. In some cases, for example, when a namespace stores groups with very large numbers of members, the shard ID is computed from both object ID and user.

3.1.2 Changelog

Zanzibar also maintains a changelog database that stores a history of tuple updates for the Watch API. The primary keys are *(changelog shard ID, timestamp, unique update ID)*, where a changelog shard is randomly selected for each write.

Every Zanzibar write is committed to both the tuple storage and the changelog shard in a single transaction. We designate the Spanner server hosting the changelog shard as the transaction coordinator to minimize blocking of changelog reads on pending transactions.

3.1.3 Namespace Config Storage

Namespace configs are stored in a database with two tables. One table contains the configs and is keyed by namespace IDs. The other is a changelog of config updates and is keyed by commit timestamps. This structure allows a Zanzibar server to load all configs upon startup and monitor the changelog to refresh configs continuously.

3.1.4 Replication

To reduce latency, Zanzibar data is replicated to be close to our clients. Replicas exist in dozens of locations around the world, with multiple replicas per region. The 5 voting replicas are in eastern and central United States, in 3 different metropolitan areas to isolate failures but within 25 milliseconds of each other so that Paxos transactions commit quickly.

3.2 Serving

3.2.1 Evaluation Timestamp

As noted in §2.4, clients can provide zookies to ensure a minimum snapshot timestamp for request evaluation. When a zookie is not provided, the server uses a default staleness chosen to ensure that all transactions are evaluated at a timestamp that is as recent as possible without impacting latency.

On each read request it makes to Spanner, Zanzibar receives a hint about whether or not the data at that timestamp required an out-of-zone read and thus incurred additional latency. Each server tracks the frequency of such out-of-zone reads for data at a default staleness as well as for fresher

and staler data, and uses these frequencies to compute a binomial proportion confidence interval of the probability that any given piece of data is available locally at each staleness.

Upon collecting enough data, the server checks to see if each staleness value has a sufficiently low probability of incurring an out-of-zone read, and thus will be low-latency. If so, it updates the default staleness bound to the lowest "safe" value. If no known staleness values are safe, we use a two-proportion z-test to see if increasing the default will be a statistically significant amount safer. In that case, we increase the default value in the hopes of improving latency. This default staleness mechanism is purely a performance optimization. It does not violate consistency semantics because Zanzibar always respects zookies when provided.

3.2.2 Config Consistency

Because changes to namespace configs can change the results of ACL evaluations, and therefore their correctness, Zanzibar chooses a single snapshot timestamp for config metadata when evaluating each client request. All `aclservers` in a cluster use that same timestamp for the same request, including for any subrequests that fan out from the original client request.

Each server independently loads namespace configs from storage continuously as they change (§3.1.3). Therefore, each server in a cluster may have access to a different range of config timestamps due to restarts or network latency. Zanzibar must pick a timestamp that is available across all of them. To facilitate this, a monitoring job tracks the timestamp range available to every server and aggregates them, reporting a globally available range to every other server. On each incoming request the server picks a time from this range, ensuring that all servers can continue serving even if they are no longer able to read from the config storage.

3.2.3 Check Evaluation

Zanzibar evaluates ACL checks by converting check requests to boolean expressions. In a simple case, when there are no userset rewrite rules, checking a user U against a userset $\langle object\#relation \rangle$ can be expressed as

$$\begin{aligned} \text{CHECK}(U, \langle object\#relation \rangle) = \\ \exists \text{ tuple } \langle object\#relation@U \rangle \\ \vee \exists \text{ tuple } \langle object\#relation@U' \rangle, \text{ where} \\ U' = \langle object'\#relation' \rangle \text{ s.t. CHECK}(U, U'). \end{aligned}$$

Finding a valid $U' = \langle object'\#relation' \rangle$ involves evaluating membership on all indirect ACLs or groups, recursively. This kind of "pointer chasing" works well for most types of ACLs and groups, but can be expensive when indirect ACLs or groups are deep or wide. §3.2.4 explains how we handle this problem. Userset rewrite rules are also translated to boolean expressions as part of check evaluation.

To minimize check latency, we evaluate all leaf nodes of the boolean expression tree concurrently. When the outcome of one node determines the result of a subtree, evaluation of other nodes in the subtree is cancelled.

Evaluation of leaf nodes usually involves reading relation tuples from databases. We apply a pooling mechanism to group reads for the same ACL check to minimize the number of read RPCs to Spanner.

3.2.4 Leopard Indexing System

Recursive pointer chasing during check evaluation has difficulty maintaining low latency with groups that are deeply nested or have a large number of child groups. For selected namespaces that exhibit such structure, Zanzibar handles checks using Leopard, a specialized index that supports efficient set computation.

A Leopard index represents a collection of named sets using (T, s, e) tuples, where T is an enum representing the set type and s and e are 64-bit integers representing the set ID and the element ID, respectively. A query evaluates an expression of union, intersection, or exclusion of named sets and returns the result set ordered by the element ID up to a specified number of results.

To index and evaluate group membership, Zanzibar represents group membership with two set types, GROUP2GROUP and MEMBER2GROUP, which we show here as functions mapping from a set ID to element IDs:

- $\text{GROUP2GROUP}(s) \rightarrow \{e\}$, where s represents an ancestor group and e represents a descendent group that is *directly or indirectly* a sub-group of the ancestor group.
- $\text{MEMBER2GROUP}(s) \rightarrow \{e\}$, where s represents an individual user and e represents a parent group in which the user is a *direct* member.

To evaluate whether user U is a member of group G, we check whether

$$(\text{MEMBER2GROUP}(U) \cap \text{GROUP2GROUP}(G)) \neq \emptyset$$

Group membership can be considered as a reachability problem in a graph, where nodes represent groups and users and edges represent direct membership. Flattening group-to-group paths allows reachability to be efficently evaluated by Leopard, though other types of denormalization can also be applied as data patterns demand.

The Leopard system consists of three discrete parts: a serving system capable of consistent and low-latency operations across sets; an offline, periodic index building system; and an online real-time layer capable of continuously updating the serving system as tuple changes occur.

Index tuples are stored as ordered lists of integers in a structure such as a skip list, thus allowing for efficient union and intersections among sets. For example, evaluating the intersection between two sets, A and B, requires only $O(\min(|A|, |B|))$ skip-list seeks. The index is sharded by element IDs and can be distributed across multiple servers. Shards are usually served entirely from memory, but they

can also be served from a mix of hot and cold data spread between memory and remote solid-state devices.

The offline index builder generates index shards from a snapshot of Zanzibar relation tuples and configs, and replicates the shards globally. It respects userset rewrite rules and recursively expands edges in an ACL graph to form Leopard index tuples. The Leopard servers continously watch for new shards and swap old shards with new ones when they become available.

The Leopard system described thus far is able to efficiently evaluate deeply and widely nested group membership, but cannot do so at a fresh and consistent snapshot due to offline index generation and shard swapping. To support consistent ACL evaluation, Leopard servers maintain an incremental layer that indexes all updates since the offline snapshot, where each update is represented by a (T, s, e, t, d) tuple, where t is the timestamp of the update and d is a deletion marker. Updates with timestamps less than or equal to the query timestamp are merged on top of the offline index during query processing.

To maintain the incremental layer, the Leopard incremental indexer calls Zanzibar's Watch API to receive a temporally ordered stream of Zanzibar tuple modifications and transforms the updates into a temporally ordered stream of Leopard tuple additions, updates, and deletions. Generating updates for the GROUP2GROUP tuples requires the incremental indexer to maintain group-to-group membership for denormalizing the effects of a relation tuple update to potentially multiple index updates.

In practice, a single Zanzibar tuple addition or deletion may yield potentially tens of thousands of discrete Leopard tuple events. Each Leopard serving instance receives the complete stream of these Zanzibar tuple changes through the Watch API. The Leopard serving system is designed to continuously ingest this stream and update its various posting lists with minimal impact to query serving.

3.2.5 Handling Hot Spots

The workload of ACL reads and checks is often bursty and subject to hot spots. For example, answering a search query requires conducting ACL checks for all candidate results, whose ACLs often share common groups or indirect ACLs. To facilitate consistency, Zanzibar avoids storage denormalization and relies only on normalized data (except for the cases described in §3.2.4). With normalized data, hot spots on common ACLs (e.g., popular groups) may overload the underlying database servers. We found the handling of hot spots to be the most critical frontier in our pursuit of low latency and high availability.

Zanzibar servers in each cluster form a distributed cache for both reads and check evaluations, including intermediate check results evaluated during pointer chasing. Cache entries are distributed across Zanzibar servers with consistent hashing [20]. To process checks or reads, we fan out requests to the corresponding Zanzibar servers via an internal RPC interface. To minimize the number of internal RPCs, for most namespaces we compute the forwarding key from the object ID, since processing a check on ⟨*object#relation*⟩ often involves indirect ACL checks on other relations of the same object and reading relation tuples of the object. These checks and reads can be processed by the same server since they share the same forwarding key with the parent check request. To handle hot forwarding keys, we cache results at both the caller and the callee of internal RPCs, effectively forming cache trees. We also use Slicer [12] to help distribute hot keys to multiple servers.

We avoid reusing results evaluated from a different snapshot by encoding snapshot timestamps in cache keys. We choose evaluation timestamps rounded up to a coarse granularity, such as one or ten seconds, while respecting staleness constraints from request zookies. This timestamp quantization allows the vast majority of recent checks and reads to be evaluated at the same timestamps and to share cache results, despite having microsecond-resolution timestamps in cache keys. It is worth noting that rounding up timestamps does not affect Zanzibar's consistency properties, since Spanner ensures that a snapshot read at timestamp T will observe all writes up to T—this holds even if T is in the future, in which case the read will wait until TrueTime has moved past T.

To handle the "cache stampede" problem [3], where concurrent requests create flash hot spots before the cache is populated with results, we maintain a *lock table* on each server to track outstanding reads and checks. Among requests sharing the same cache key only one request will begin processing; the rest block until the cache is populated.

We can effectively handle the vast majority of hot spots with distributed caches and lock tables. Over time we made the following two improvements.

First, direct membership checks of a user for an object and relation (i.e. ⟨*object#relation@user*⟩) are usually handled by a single relation tuple lookup. However, occasionally a very popular object invites many concurrent checks for different users, causing a hot spot on the storage server hosting relation tuples for the object. To avoid these hot spots, we read and cache *all* relation tuples of ⟨*object#relation*⟩ for the hot object, trading read bandwidth for cacheability. We dynamically detect hot objects to apply this method to by tracking the number of outstanding reads on each object.

Second, indirect ACL checks are frequently cancelled when the result of the parent ACL check is already determined. This leaves the cache key unpopulated. While eager cancellation reduces resource usage significantly, it negatively affects latency of concurrent requests that are blocked by the lock table entry. To prevent this latency impact, we delay eager cancellation when there are waiters on the corresponding lock table entry.

3.2.6 Performance Isolation

Performance isolation is indispensable for shared services targeting low latency and high availability. If Zanzibar or one of its clients occasionally fails to provision enough resources to handle an unexpected usage pattern, the following isolation mechanisms ensure that performance problems are isolated to the problematic use case and do not adversely affect other clients.

First, to ensure proper allocation of CPU capacity, Zanzibar measures the cost of each RPC in terms of generic *cpu-seconds*, a hardware-agnostic metric. Each client has a global limit on maximum CPU usage per second; its RPCs will be throttled if it exceeds the limit *and* there is no spare capacity in the overall system.

Each Zanzibar server also limits the total number of outstanding RPCs to control its memory usage. Likewise it limits the number of oustanding RPCs per client.

Zanzibar further limits the maximum number of concurrent reads per *(object, client)* and per client on each Spanner server. This ensures that no single object or client can monopolize a Spanner server.

Finally, we use different lock table keys for requests from different clients to prevent any throttling that Spanner applies to one client from affecting other clients.

3.2.7 Tail Latency Mitigation

Zanzibar's distributed processing requires measures to accommodate slow tasks. For calls to Spanner and to the Leopard index we rely on request hedging [16] (i.e. we send the same request to multiple servers, use whichever response comes back first, and cancel the other requests). To reduce round-trip times, we try to place at least two replicas of these backend services in every geographical region where we have Zanzibar servers. To avoid unnecessarily multiplying load, we first send one request and defer sending hedged requests until the initial request is known to be slow.

To determine the appropriate hedging delay threshold, each server maintains a delay estimator that dynamically computes an Nth percentile latency based on recent measurements. This mechanism allows us to limit the additional traffic incurred by hedging to a small fraction of total traffic.

Effective hedging requires the requests to have similar costs. In the case of Zanzibar's authorization checks, some checks are inherently more time-consuming than others because they require more work. Hedging check requests would result in duplicating the most expensive workloads and, ironically, worsening latency. Therefore we do not hedge requests between Zanzibar servers, but rely on the previously discussed sharding among multiple replicas and on monitoring mechanisms to detect and avoid slow servers.

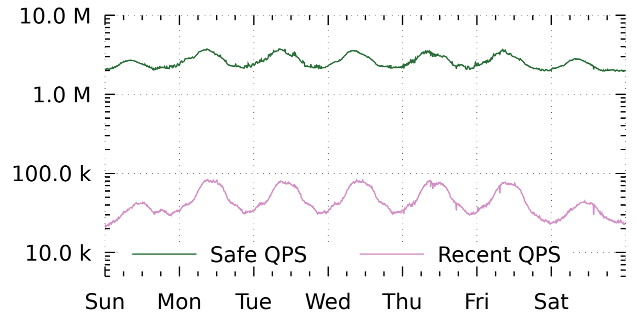


Figure 3: Rate of Check Safe and Check Recent requests over a 7-day period in December 2018.

4 Experience

Zanzibar has been in production use for more than 5 years. Throughout that time, the number of clients using Zanzibar and the load they place on Zanzibar have grown steadily. This section discusses our experience operating Zanzibar as a globally distributed authorization system.

Zanzibar manages more than 1,500 namespaces defined by hundreds of client applications. The size of a namespace configuration file serves as a rough measure of the complexity of the access control policy implemented by that namespace. These configuration files range from tens of lines to thousands of lines, with the median near 500 lines.

These namespaces contain more than 2 trillion relation tuples that occupy close to 100 terabytes. The number of tuples per namespace ranges over many orders of magnitude, from tens to a trillion, with the median near 15,000. This data is fully replicated in more than 30 locations around the world to maintain both proximity to users and high availability.

Zanzibar serves more than 10 million client queries per second (QPS). Over a sample 7-day period in December 2018, Check requests peak at roughly 4.2M QPS, Read at 8.2M, Expand at 760K, and Write at 25K. Queries that read data are thus two orders of magnitude more frequent than those that write data.

Zanzibar distributes this load across more than 10,000 servers organized in several dozen clusters around the world. The number of servers per cluster ranges from fewer than 100 to more than 1,000, with the median near 500. Clusters are sized in proportion to load in their geographic regions.

4.1 Requests

We divide requests into two categories according to the required data freshness, which can have a large impact on latency and availability of the requests. Specifically, Check, Read, and Expand requests carry zookies to specify lower bounds on evaluation timestamps. When a zookie timestamp is higher than that of the most recent data replicated to the region, the storage reads require cross-region round trips to the leader replica to retrieve fresher data. As our storage

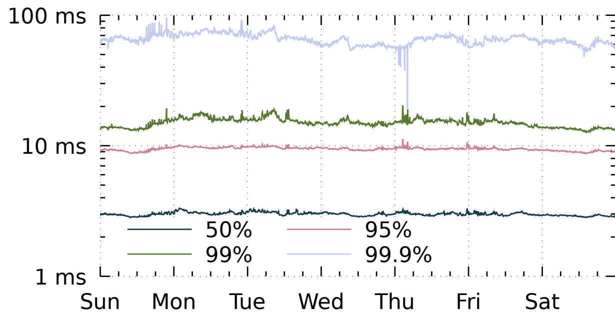


Figure 4: Latency of Check Safe responses at different percentiles over a 7-day period in December 2018.

is configured with replication heartbeats with 8-second intervals, we divide our requests into two categories: *Safe* requests have zookies more than 10 seconds old and can be served within the region most of time, while *Recent* requests have zookies less than 10 seconds old and often require inter-region round trips. We report separate statistics for each.

Figure 3 shows the rate of Check Safe and Check Recent requests over 7 days. Both exhibit a diurnal cycle. The rate of Safe requests is about two orders of magnitude larger than that of Recent requests, which allows Zanzibar to serve the vast majority of ACL checks locally.

4.2 Latency

Zanzibar's latency budget is generally a small fraction of the few hundreds of milliseconds of total response time that its clients must provide to be viable interactive services. Consider for example a client that performs authorization checks on multiple documents before it can show the results of a search on those documents.

We measure latency on the server side using live traffic because (1) latency is heavily influenced by our caching and de-duplication mechanisms so that it is only realistically reflected by live traffic, and (2) accurately measuring latency from clients requires well-behaving clients. Provisioning of client jobs is outside of Zanzibar's control and sometimes client jobs are overloaded.

Figure 4 shows the latency of Check Safe responses over 7 days. At the 50th, 95th, 99th, and 99.9th percentiles it peaks at roughly 3, 11, 20, and 93 msec, respectively. This performance meets our latency goals for an operation that is frequently in the critical path of user interactions.

Table 2 summarizes the latency distributions of Check, Read, Expand, and Write responses over the same 7 days. As intended, the more frequently used Safe versions of Check, Read, and Expand are significantly faster than the less frequently used Recent versions. Writes are the least frequently used of all the APIs, and the slowest because they always require distributed coordination among Spanner servers.

		Latency in milliseconds, μ (σ)		
	API	**50%ile**	**95%ile**	**99%ile**
Safe	Check	3.0 (0.091)	9.46 (0.3)	15.0 (1.19)
	Read	2.18 (0.031)	3.71 (0.094)	8.03 (3.28)
	Expand	4.27 (0.313)	8.84 (0.586)	34.1 (4.35)
Recent	Check	2.86 (0.087)	60.0 (2.1)	76.3 (2.59)
	Read	2.21 (0.054)	40.1 (2.03)	86.2 (3.84)
	Expand	5.79 (0.224)	45.6 (3.44)	121.0 (2.38)
	Write	127.0 (3.65)	233.0 (23.0)	401.0 (133.0)

Table 2: Mean and standard deviation of RPC response latency over a 7-day period in December 2018.

4.3 Availability

We define availability as the fraction of "qualified" RPCs the service answers successfully within latency thresholds: 5 seconds for a Safe request, and 15 seconds for a Recent request as leader re-election in Spanner may take up to 10 seconds. For an RPC to be qualified, the request must be well-formed and have a deadline longer than the latency threshold. In addition, the client must stay within its resource quota.

For these reasons, we cannot measure availability directly with live traffic, as our clients sometimes send RPCs with short deadlines or cancel their in-progress RPCs. Instead, we sample a small fraction of valid requests from live traffic and replay them later with our own probers. When replaying the requests, we set the timeout to be longer than the availability threshold. We also adjust the request zookie, if one is specified, so that the relative age of the zookie remains the same as when the request was received in the live traffic. Finally, we run 3 probers per cluster and exclude outliers to eliminate false alarms caused by rare prober failures.

To compute availability, we aggregate success ratios over 90-day windows averaged across clusters. Figure 5 shows Zanzibar's availability as measured by these probers. Availability has remained above 99.999% over the past 3 years of operation at Google. In other words, for every quarter, Zanzibar has less than 2 minutes of global downtime and fewer than 13 minutes when the global error ratio exceeds 10%.

4.4 Internals

Zanzibar servers delegate checks and reads to each other based on consistent hashing, and both the caller and the callee sides of the delegated operations cache the results to prevent hot spots (§3.2.5). At peak, Zanzibar handles 22 million internal "delegated" RPCs per second, split about evenly between reads and checks. In-memory caching handles approximately 200 million lookups per second at peak, 150 million from checks and 50 million from reads. Caching for

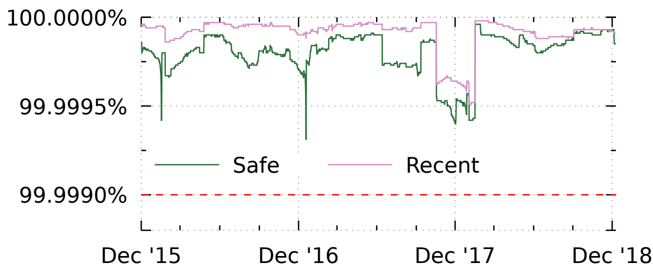


Figure 5: Zanzibar's availability over the past three years has remained above 99.999%.

checks has a 10% hit rate on the delegate's side, with an additional 12% saved by the lock table. Meanwhile, caching on the delegator's side has a 2% hit rate with an additional 3% from the lock table. While these hit rates appear low, they prevent 500K internal RPCs per second from creating hot spots.

Delegated reads see higher hit rates on the delegate's side—24% on the cache and 9% on the lock table—but the delegator's cache is hit less than 1% of the time. For super-hot groups, Zanzibar further optimizes by reading and caching the full set of members in advance—this happens for 0.1% of groups but further prevents hot spots.

This caching, along with aggressive pooling of read requests, allows Zanzibar to issue only 20 million read RPCs per second to Spanner. The median of these requests reads 1.5 rows per RPC, but at the 99th percentile they each read close to 1 thousand rows.

Zanzibar's Spanner reads take 0.5 msec at the median, and 2 msec at the 95th percentile. We find that 1% of Spanner reads, or 200K reads per second, benefit from hedging. We note that Zanzibar uses an instance of Spanner that runs internally to Google, not an instance of Cloud Spanner [6].

The Leopard index is performing 1.56M QPS at the median, or 2.22M QPS at the 99th percentile, based on data aggregated over 7 days. Over the same 7 days, Leopard servers respond in fewer than 150 µsec at the median, or under 1 msec at the 99th percentile. Leopard's incremental layer during those 7 days writes roughly 500 index updates per second at the median, and approximately 1.5K updates per second at the 99th percentile.

4.5 Lessons Learned

Zanzibar has evolved to meet the varied and heavy demands of a growing set of clients, including Google Calendar, Google Cloud, Google Drive, Google Maps, Google Photos, and YouTube. This section highlights lessons learned from this experience.

One common theme has been the importance of flexibility to accommodate differences between clients. For example:

- *Access control patterns vary widely:* Over time we have added features to support specific clients. For instance, we added `computed_userset` to allow inferring an object's owner ID from the object ID prefix, which reduces space requirements for clients such as Drive and Photos that manage many private objects. Similarly, we added `tuple_to_userset` to represent object hierarchy with only one relation tuple per hop. The benefits are both space reduction and flexibility—it allows clients such as Cloud both to express ACL inheritance compactly and to change ACL inheritance rules without having to update large numbers of tuples. See §2.3.1.
- *Freshness requirements are often but not always loose:* Clients often allow unspecified, moderate staleness during ACL evaluation, but sometimes require more precisely specified freshness. We designed our zookie protocol around this property so that we can serve most requests from a default, already replicated snapshot, while allowing clients to bound the staleness when needed. We also tuned the granularity of our snapshot timestamps to match clients' freshness requirements. The resulting coarse timestamp quanta allow us to perform the majority of authorization checks on a small number of snapshots, thus greatly reducing the frequency of database reads. See §3.2.1.

Another theme has been the need to add performance optimizations to support client behaviors observed in production. For example:

- *Request hedging is key to reducing tail latency:* Clients that offer search capabilities to their users, such as Drive, often issue tens to hundreds of authorization checks to serve a single set of search results. We introduced hedging of Spanner and Leopard requests to prevent an occasional slow operation from slowing the overall user interaction. See §3.2.7.
- *Hot-spot mitigation is critical for high availability:* Some workloads create hot spots in ACL data that can overwhelm the underlying database servers. A common pattern is a burst of ACL checks for an object that is indirectly referenced by the ACLs for many different objects. Specific instances arise from the search use case mentioned above, where the documents in the search indirectly share ACLs for a large social or work group, and Cloud use cases where many objects indirectly share ACLs for the same object high in a hierarchy. Zanzibar handles most hot spots with general mechanisms such as its distributed cache and lock table, but we have found the need to optimize specific uses cases. For example, we added cache prefetching of all relation tuples for a hot object. We also delayed cancellation of secondary ACL checks when there are concurrent requests for the same ACL data. See §3.2.5.
- *Performance isolation is indispensable to protect against misbehaving clients:* Even with hot-spot mitigation measures, unexpected and sometimes unin-

tended client behaviors could still overload our system or its underlying infrastructure. Examples include when clients launch new features that prove unexpectedly popular or exercise Zanzibar in unintended ways. Over time we have added isolation safeguards to ensure that there are no cascading failures between clients or between objects of the same client. These safeguards include fine-grained cost accounting, quotas, and throttling. See §3.2.6.

5 Related Work

Zanzibar is a planet-scale distributed ACL storage and evaluation system. Many of its authorization concepts have been explored previously within the domains of access control and social graphs, and its scaling challenges have been investigated within the field of distributed systems.

Access control is a core part of multi-user operating systems. Multics [23] supports ACLs on segments and directories. ACL entries consist of a principal identifier and a set of permissions bits. In the first edition of UNIX [9], file flags indicate whether owner and non-owner can read or write the file. By the 4th edition, the permissions bits had been expanded to read/write/execute bits for owner, group, and others. POSIX ACLs [4] add an arbitrary list of users and groups, each with up to 32 permissions bits. VMS [7, 8] supports ACL inheritance for files created within a directory tree. Zanzibar's data model supports permissions, users, groups, and inheritance as found in the above systems.

Taos [24, 10] supports compound principals that incorporate how an identity has been transformed as it passes through a distributed system. For example, if user U logged into workstation W to access file server S, S would see requests authenticated as "W for U" rather than just U. This would allow one to write an ACL on a user's e-mail that would be accessible only to the user, and only if being accessed via the mail server. Abadi et al. discuss in [11] a model of group-based ACLs with support for compound identities. Their notion of "blessings" are similar to Zanzibar tuples. However, Zanzibar adopts a unified representation for ACLs and groups using usersets, while they are separate concepts in [11].

Role-based access control (RBAC), first proposed in [17], introduced the notion of *roles*, which are similar to Zanzibar relations. Roles can inherit from each other and imply permissions. A number of Zanzibar clients have implemented RBAC policies on top of Zanzibar's namespace configuration language.

A discussion of ACL stores in 2019 would be remiss without mentioning the Identity and Access Management (IAM) systems offered commercially by Amazon [1], Google [5], Microsoft [2], and others. These systems allow customers of those companies' cloud products to configure flexible access controls based on various features such as: assigning users to roles or groups; domain-specific policy languages; and APIs that allow the creation and modification of ACLs. What all of these systems have in common is unified ACL storage and an RPC-based API, a philosophy also core to Zanzibar's design. Google's Cloud IAM system [5] is built as a layer on top of Zanzibar's ACL storage and evaluation system.

TAO [13] is a distributed datastore for Facebook's social graph. Several Zanzibar clients also use Zanzibar to store their social graphs. Both Zanzibar and TAO provide authorization checks to clients. Both are deployed as single-instance services, both operate at a large scale, and both are optimized for read-only operations. TAO offers eventual global consistency with asynchronous replication and best-effort read-after-write consistency with synchronous cache updates. In contrast, Zanzibar provides external consistency and snapshot reads with bounded staleness, so that it respects causal ordering between ACL and content updates and thus protects against the "new enemy" problem.

Lamport clocks [22] provide partially ordered vector timestamps that can be used to determine the order of events. However, Lamport clocks require explicit participation of all "processes", where in Zanzibar's use cases some of the "processes" can be external clients or even human users. In contrast, Zanzibar relies on its underlying database system, Spanner [15], to offer both external consistency and snapshot reads with bounded staleness. In particular, Zanzibar builds on Spanner's TrueTime abstraction [15] to provide linearizable commit timestamps encoded as zookies.

At the same time, Zanzibar adds a number of features on top of those provided by Spanner. For one, the zookie protocol does *not* let clients read or evaluate ACLs at an arbitrary snapshot. This restriction allows Zanzibar to choose a snapshot that facilitates fast ACL evaluation. In addition, Zanzibar provides resilience to database hotspots (e.g. authorization checks on a suddenly popular video) and safe pointer chasing despite potentially deep recursion (e.g. membership checks on hierarchical groups).

The Chubby distributed lock service [14] offers reliable storage, linearizes writes, and provides access control, but it lacks features needed to support Zanzibar's use cases. In particular, it does not support high volumes of data, efficient range reads, or reads at a client-specified snapshot with bounded staleness. Its cache invalidation mechanism also limits its write throughput.

Finally, ZooKeeper offers a high-performance coordination service [19] but also lacks features required by Zanzibar. Relative to Chubby, it can handle higher read and write rates with more relaxed cache consistency. However, it does not provide external consistency for updates across different nodes since its linearizability is on a per-node basis. It also does not provide snapshot reads with bounded staleness.

6 Conclusion

The Zanzibar authorization system unifies access control data and logic for Google. Its simple yet flexible data model and configuration language support a variety of access control policies from both consumer and enterprise applications.

Zanzibar's external consistency model is one of its most salient features. It respects the ordering of user actions, yet at the same time allows authorization checks to be evaluated at distributed locations without global synchronization.

Zanzibar employs other key techniques to provide scalability, low latency, and high availability. For example, it evaluates deeply or widely nested group membership with Leopard, a specialized index for efficient computation of set operations with snapshot consistency. As another example, it combines a distributed cache with a mechanism to deduplicate in-flight requests. It thus mitigates hot spots, a critical production issue when serving data on top of normalized, consistent storage. These measures together result in a system that scales to trillions of access control rules and millions of authorization requests per second.

7 Acknowledgments

Many people have made technical contributions to Zanzibar. We thank previous and recent members of the development team, including Dan Barella, Miles Chaston, Daria Jung, Alex Mendes da Costa, Xin Pan, Scott Smith, Matthew Steffen, Riva Tropp, and Yuliya Zabiyaka. We also thank previous and current members of the Site Reliability Engineering team, including Randall Bosetti, Hannes Eder, Robert Geisberger, Tom Li, Massimo Maggi, Igor Oks, Aaron Peterson, and Andrea Yu.

In addition, a number of people have helped to improve this paper. We received insightful comments from David Bacon, Carolin Gäthke, Brad Krueger, Ari Shamash, Kai Shen, and Lawrence You. We are also grateful to Nadav Eiron and Royal Hansen for their support. Finally, we thank the anonymous reviewers and our shepherd, Eric Eide, for their constructive feedback.

References

[1] Amazon Web Services Identity and Access Management. `https://aws.amazon.com/iam/`. Accessed: 2019-04-16.

[2] Azure Identity and Access Management. `https://www.microsoft.com/en-us/cloud-platform/identity-management`. Accessed: 2019-04-16.

[3] Cache stampede. `https://en.wikipedia.org/wiki/Cache_stampede`. Accessed: 2019-04-16.

[4] DCE 1.1: Authentication and Security Services. `http://pubs.opengroup.org/onlinepubs/9668899`. Accessed: 2019-04-16.

[5] Google Cloud Identity and Access Management. `https://cloud.google.com/iam/`. Accessed: 2019-04-16.

[6] Google Cloud Spanner. `https://cloud.google.com/spanner/`. Accessed: 2019-04-16.

[7] HP OpenVMS System Management Utilities Reference Manual. `https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04622366`. Accessed: 2019-04-16.

[8] OpenVMS Guide to System Security. `http://www.itec.suny.edu/scsys/vms/ovmsdoc073/V73/6346/6346pro_006.html#acl_details`. Accessed: 2019-04-16.

[9] Unix Manual. `https://www.bell-labs.com/usr/dmr/www/pdfs/man22.pdf`. Accessed: 2019-04-16.

[10] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept. 1993), 706–734.

[11] ABADI, M., BURROWS, M., PUCHA, H., SADOVSKY, A., SHANKAR, A., AND TALY, A. Distributed authorization with distributed grammars. In *Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security - Volume 9465* (New York, NY, USA, 2015), Springer-Verlag New York, Inc., pp. 10–26.

[12] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 739–753.

[13] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC '13, pp. 49–60.

[14] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the*

7th Symposium on Operating Systems Design and Implementation (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[15] 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., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI '12, pp. 251–264.

[16] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[17] FERRAIOLO, D., AND KUHN, R. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference* (1992), pp. 554–563.

[18] GIFFORD, D. K. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[19] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association.

[20] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. 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* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[21] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[22] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[23] SALTZER, J. H. Protection and control of information sharing in Multics. In *Proceedings of the Fourth ACM Symposium on Operating System Principles* (New York, NY, USA, 1973), SOSP '73, ACM.

[24] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 256–269.

IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services

Biswaranjan Panda, Deepthi Srinivasan, Huan Ke*,
Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi*

Nutanix Inc. University of Chicago*

Abstract

We address the problem of "fail-slow" fault, a fault where a hardware or software component can still function (does not fail-stop) but in much lower performance than expected. To address this, we built IASO, a peer-based, non-intrusive fail-slow detection framework that has been deployed for more than 1.5 years across 39,000 nodes in our customer sites and helped our customers reduce major outages due to fail-slow incidents. IASO primarily works based on timeout signals (a negligible overhead of monitoring) and converts them into a stable and accurate fail-slow metric. IASO can quickly and accurately isolate a slow node within minutes. Within a 7-month period, IASO managed to catch 232 fail-slow incidents in our large deployment field. In this paper, we have also assembled a large dataset of 232 fail-slow incidents along with our analysis. We found that the fail-slow annual failure rate in our field is 1.02%.

1 Introduction

Maintaining high availability of distributed storage services in real deployment fields is challenging due to the various types of faults that can occur. In the last few years, there has been an emphasis on "fail-slow" fault mode [28, 32]. This means that a hardware or software component can still function (does not fail-stop) but in much lower performance than expected. Such faults have been studied under different names such "gray failure" [32], "limping" [24, 37], and "partial failures" [29]. We chose the term "fail-slow" for simplicity and reflecting a recent term [28].

The urgency here is that many distributed systems are still designed based on a binary model of no failure and fail-stop failures. Recent works shows that many distributed systems cannot gracefully tolerate fail-slow mode, i.e. the system cannot isolate and hide a fail-slow component, causing latency spikes or throughput degradation to users [24, 28, 31, 32, 56]. Worse, it has been reported that a fail-slow component can cause *cascade* of performance failures across the cluster, bringing down services for hours [24, 28]. This calls for the importance of designing systems that tolerate not just absolute failure of sub-components but can also gracefully handle the occurrence of performance faults.

In this context, our work in this paper makes the two following contributions:

(1) Design and implementation of a fail-slow mitigation framework. The first contribution of the paper is IASO, our peer-based, non-intrusive fail-slow detection framework that has been deployed for more than 1.5 years across 39,000 nodes in our customer sites. Before the integration with IASO we had more than 25 full outages (IOPS went to zero) due to cascading impacts of fail-slow incidents, not to mention many other occurrences of partial slowdowns. Since the integration with IASO, we had only 2 major outages (false negative cases) caused by fail slow.

Motivation: IASO is motivated by the following reasons.

First, we found that fail-slow faults can be caused by many root causes. Sole dependence on low-level detection tools [38, 40, 15, 4] at various levels of the software and hardware stack might not be sufficient. Thus, we need a fail-slow detection system that works at the service (distributed system) level. Most existing work focuses on hardware level outlier detection or software performance bugs but they might not cover all of the detailed root causes occurring in the field (§4.2.3).

Second, most existing efforts focus only on detection but not mitigation. We are only aware of a handful of works that perform mitigation in real deployments (more in §5). Yet, our findings suggest that if fail-slow incidents are not quickly and automatically isolated, it can cascade and directly affect users for hours or days. For this reason, it is paramount that deployed systems are equipped with fail-slow mitigation.

Third, although some computing frameworks such as MapReduce [1, 23] are equipped with fail-slow mitigation (*e.g.*, via speculative execution [58] or cloning [10]), the tail tolerance is built in their abstractions (*e.g.*, "jobs", "tasks") and not directly generalizable to many other distributed systems. Recent works revealed that many other distributed systems are still not fail-slow tolerant [24, Figure 1][56, Figure 12]. Hence, we need a more general way of addressing fail-slow faults in many distributed storage services.

Challenges and solutions: A fail-slow detection framework must be non-intrusive (negligible overhead), stable and accurate, and not accidentally make wrong decisions (*e.g.*, quarantine healthy nodes). To achieve this, we make IASO peer-based, *i.e.*, a slow service instance should be compared against its peers of the same service (*e.g.*, the performance of Cassandra instance should not be compared to ZooKeeper's). We also make IASO load aware, *i.e.*, the relative performance of a service instance must not be improved or worsen just because the load on the node on which the instance is running on is different.

To achieve all of these, we created an algorithm (§2.2) that can work solely based on timeout signals. Our algorithm can convert timeout and successful-response statistics into a stable and accurate fail-slow detector. Our framework does not need to monitor every request latency, hence achieving a neglibible overhead. IASO can quickly and accurately isolate a slow node within minutes. Within a 7-month period, IASO managed to catch 232 fail-slow incidents in our large deployment field. IASO also automatically quarantined the slow nodes and restored the clusters back to a healthier performance. We only encountered 9 confirmed false positives. Other false positives are because the fail-slowness disappeared when our engineers started diagnosing them (*e.g.*, perhaps caused by unknown external conditions).

(2) A dataset and analysis of fail-slow incidents With IASO integration, we were able to capture many fail-slow incidents in the field. We have assembled a large dataset of fail-slow incidents along with our analysis [7]. To the best of our knowledge, this is the largest dataset of fail-slow cases publicly reported from within a company. Furthermore, existing accounts of fail-slow accidents are anecdotal [12, 28, 32], while our contribution includes some quantitative analysis (*e.g.*, AFR, age correlation).

The dataset: The dataset contains 232 validated cases collected from the deployment of 39,000 nodes throughout a period of 7 months.[1] This data pertains to a type of fully hyperconverged system [9] that we deploy in customer sites.

Findings: Our rich dataset allows us to make some statistical findings. First, given 232 independent cases across 39,000 nodes over 7 months, we can derive that the annual failure rate is 1.02% (232 × 12 / 7 / 39,000), which is relatively significant compared to rates of other types of faults (§4.2.1). Second, we uncovered a wide range of root causes (and the low-level sub-causes), which again accentuates the need for detection at the service level, not just at the individual hardware level. Third, we also observed the "infant mortality" pattern where younger machines exhibit more fail-slow incidents. Fourth, we show that if not mitigated properly, fail-slow cases can take hours or days to fully resolve, which again highlights the importance of automatically quarantining slow nodes.

The following sections present the design and implementation of IASO (§2), experimental results (§3), our dataset and findings (§4), related work and conclusion.

2 IASO

This section presents IASO, our framework for detecting the presence of an unhealthy node and enabling self healing of the cluster. We name our system after "Iaso", the Greek goddess of recuperation from illness [8]. IASO is comprised of three stages:

1. *Detection (§2.1-2.2):* This step reduces the time to detect fail-slow incidents from hours to minutes while keeping false positives low.
2. *Mitigation (§2.3):* This step quarantines the faulty node and brings the cluster back to operation.
3. *Resolution (§2.4):* IASO automatically pages site reliability engineers (SREs) to identify the failed component and help support to do breakfix and assimilate the fixed component back into operation.

When building IASO, we adhere to the following design principles.

- *Non intrusive*: We attempt to reach a near 0% overhead, hence we use raw metrics that the deployed services already collect (*e.g.*, number of timeouts and successful responses).
- *Peer based:* A slow service instance should be compared against its peers of the same service, *e.g.*, the performance of Cassandra instance should be compared to other Cassandra instances, not ZooKeeper instances, as different types of services observe different types of workload. For this reason, we monitor at service-level requests, not at OS or hardware level.
- *Load aware:* The slowdown detection system must be aware of the service load. The relative performance of a peer must not be improved or worsen just because the load on the node the peer is running on is different. This means that the performance of a node must be normalized based on the capacity of the node; in our deployment, a cluster can have different machine capacities with different loads.
- *Stable and accurate:* As a degraded node will be quarantined, it is important to have a stable and accurate algorithm that does not accidentally make wrong decisions (false positives).

[1] For this publication we only have analyzed the dataset for a 7 month period in 2017. Data from 2018 is still being perused and cleaned.

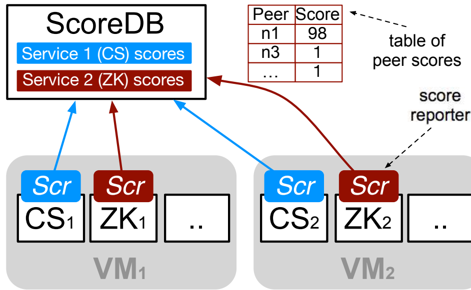


Figure 1: **IASO components.** *The figure is described in the last paragraph of page 2 and also in Section 2.1. "Scr" denotes the hook that sends score table to ScoreDB.*

The following are the terms we use in this paper. As shown in Figure 1, our *system* S is a cluster of high-end *machines* (gray shades) running *VMs* wherein *services* are running (boxes). For example, S comprises a ZooKeeper (ZK) service for cluster configuration manager, a Cassandra service (CS) for storing metadata, and our own blob-store service for storing data. Each VM runs an instance of each of the services (*e.g.*, a VM runs three service instances, Cassandra, ZooKeeper and blob-store instances). These VMs are also known as controller VMs.

2.1 Detection

Our first goal is to detect which service instance is experiencing a slowdown. Currently we only address persistent fault, *i.e.*, the instance is not being slowed down due to an intermittent condition such as a one-off high GC time. This section describes the main components of IASO as shown in Figure 1. The next section (§2.2) presents the detailed algorithm.

RAW METRICS (LATENCY VS. TIMEOUTS): One naive method to measure degradation is to measure the latency of every request. However, with today's high-throughput services it is not amenable (*e.g.*, per-node Cassandra throughput can reach 20,000 IOPS [5]). Sampling can be a solution, but we explored a different method.

In this work, we try a much cheaper method to detect degradation: counting timeouts. Many services such as Cassandra already have a built-in metric that collects how many responses were successful as well as the failed ones due to timeouts. Another advantage of using timeouts is that our monitoring system is not intrusive to the performance of the service itself (a nearly 0% overhead as counting timeouts and successful responses is a simple increment operation).

SCORES: We found that using raw timeout counts as a direct metric to measure outlier is not a stable and accurate way. Thus, we need to introduce the concept of *"score"*. Given a cluster of N nodes with N instances of a service, every instance can observe the performance of its $N-1$ peers and maintain a "score table" (as shown in Figure 1).

STABLE SCORES: The primary challenge we address in this work is *how to convert timeout and success statistics into a stable and accurate degradation detector*. Noisy scores can lead to more false positives where healthy nodes might be accidentally removed and vice versa. Later, the experiment section shows other unsuccessful algorithms that we tried (§3) which then led us to the current algorithm (§2.2). One key to prevent scores from fluctuating along with the number of timeouts is by incorporating additive increase and multiplicative decrease (AIMD) [18] such as used in TCP congestion avoidance. Thus, our custom algorithm employs a technique similar to AIMD.

SCOREDB SERVER: The scores collected from the service instances are stored in a database server called ScoreDB (Figure 1). For every peer, every instance keeps a score, hence in total ScoreDB maintains $N\times(N-1)$ score variables (per every service monitored) including their historical values. Given these scores, ScoreDB runs an outlier detection part of our algorithm and quarantines the outlier. ScoreDB is also a replicated system (to anticipate degradation within itself).

2.2 Detection Algorithm

We now describe how IASO calculates the score metric and detects an outlier. The challenge is to convert timeout and success statistics into a stable and accurate degradation detector. For every equation listed below, the explanation is in the paragraph preceding the equation. Symbols † and ‡ are used for backward references.

2.2.1 Peer Scores

Given a cluster of N instances within a service (*e.g.*, Cassandra), every instance observes the performance of its peers and puts the corresponding scores in a score table containing $N-1$ peer scores. In our scoring system below, a score ranges from 1 to 100 where a higher value implies more severe degradation. For example, in the score table in Figure 1, Cassandra instance on `Node2` believes that Cassandra instane on `Node1` is slow (a score of 98).

As score continues to change, below we use `prev` and `score` to represent the scores in the last and current epoch respectively. An epoch is the interval at which every service instance runs the equations below (*i.e.*, calculates a new score). The `epoch` is set to be 5 seconds and `prev` to 1 in the beginning.

Next, we introduce `ToRespRatio`, the ratio of the number of timeouts and total responses between two peers within an epoch. This is essential to the load-awareness part of our algorithm, that timeout counts should be relative to the num-

ber of total responses as the number of responses will vary across peers.

$\texttt{ToRespRatio} = \texttt{\#timeouts} \, / \, \texttt{\#responses}$

We then set `ratioThresh`, a timeout-response ratio threshold, with a constant value of 0.1 (*e.g.*, 10 timeouts for every 100 responses). In our experience, 10% timeouts from a peer can cause a whole-cluster degradation. A higher value may make IASO react too late, while a lower may lead to more false positives (*i.e.*, too sensitive). If `ToRespRatio` is larger than the `ratioThresh`, it is likely a heavy degradation. Otherwise, it is likely caused by a temporary high load or a benign cause.

$\texttt{ratioThresh} = 0.1$

Next, we introduce `minTTR` as the minimum time to observe zero timeout from a peer before the score assigned to it decreases from 100 to 1 (slow to healthy). We set the time to be 2 minutes. The idea is that when a peer exhibits a zero timeout, it might mean that this peer is temporarily healthy but might suffer degradation again soon. The 2-minute mark is the time window in which a peer must "prove" itself that it is really healthy. A smaller window increases the risk that we may start assigning good scores to a temporarily good peer and thereby creating an unstable score pattern. A larger window has the disadvantage that we may mark a peer as fail-slow even if it has just completely recovered but the 2-minute window hasn't passed. However, the latter scenario should be infrequent.

$\texttt{minTTR} = 2 \text{ mins}$

With all of the values above, now we can stitch them into the score calculation. In every epoch, if `ToRespRatio` is 0 (no timeout), then the `score` will be calculated as shown below. This is the "additive decrease" part of our algorithm – the score will be slowly decreasing back to zero to show that the peer is really healthy.

[if `ToRespRatio` is 0]
$\texttt{score} = \texttt{prev} - (\, 100 \times \texttt{epoch} \, / \, \texttt{minTTR} \,)$

Now, we discuss the case where some timeouts are observed (`ToRespRatio` is not zero). First, we introduce `minRatio` as a higher bound of the timeout-response ratio and threshold values. The idea here is that `ToRespRatio` can be very high (*e.g.*, 90%, when a peer is highly unresponsive). This high value will make our algorithm below unstable. Thus, we cap it to the `ratioThresh` value (0.1), *i.e.*, 10% already represents enough degradation.

$\texttt{minRatio} = \min \left(\texttt{ToRespRatio} \, , \texttt{ratioThresh} \right)^{\dagger}$

Finally, the last variable we introduce is `nearThresh` to measure how far the timeout-response ratio to the threshold (how far from the 10% timeouts). This threshold nearness ranges from 0 to 1.0.

$\texttt{nearThresh} = \texttt{minRatio} \, / \, \texttt{ratioThresh} \; ^{\ddagger}$

With all the new variables above, we now can calculate the score when timeout-response ratio is higher than zero. The equation below represents the "multiplicative increase" part of our algorithm where the score is increased by the threshold nearness. We put more examples below.

[if `ToRespRatio` is not 0]
$\texttt{score} = \texttt{prev} + (\, \texttt{prev} \times \texttt{nearThresh} \,)$

Let's use an example where an instance gave a score of 32 for a peer instance in the last epoch. Now, the current epoch sees too many timeouts beyond the threshold such that `nearThresh` is 1.0. Thus, the current score will jump from 32 to 32+32 (*i.e.*, the score increases multiplicatively).[2]

$\texttt{score} = 32 + (\, 32 \times 1.0 \,) = 64$

Let's imagine another scenario where the `ToRespRatio` is as small as 0.01 (1% timeouts) . Here, the `minRatio` will be 0.01 (see equation †) and the `nearThresh` be 0.1 (see ‡). Thus, the next `score` will only increment fractionally:

$\texttt{score} = 32 + (\, 32 \times 0.1 \,) = 35.2$

To sum up, our algorithm prevents scores from fluctuating along with the number of timeouts. That is, we linearly decrement the score when we do not observe any timeouts from a peer, but multiplicatively increase the score when we observe timeouts from the peer.

2.2.2 Scores Set

Every instance X then sends the scores of its peers (A, B, ...) to the ScoreDB server, which will then maintain a history of the scores. For example for a given peer A, there are $N-1$ scores for A collected in every 5-second epoch.

For every peer, all the scores given for that peer are collected within a 10-minute sliding window, where ScoreDB then picks the 30th-percentile value to be the *representative* score for that peer, such as for instance A. The 30th-percentile value implies that the peer instance must have 70% high score values within a 10-minute interval such that we do not inadvertently quarantine instances with mere transient faults. In our deployments, we have observed that a 10-minute window is wide enough to detect persistent faults. It may not be the absolute minimum but it does put an upper bound on the time to isolate a fail-slow peer.

At this point, ScoreDB has N representative scores for all the instances in the cluster and it submits these scores to the DBSCAN algorithm [6]. ScoreDB performs this every minute, but using the data from the past 10 minutes (a sliding window). DBSCAN [6] is an algorithm that takes a set of

[2] To make the score multiplication increases faster/slower (*i.e.*, more configurable), we can introduce a score multiplier with a usage such as: *e.g.*, 32 + `scoreMultiplier` × 32. We use `scoreMultiplier` =1.

points and groups them such that points that are spatially close are grouped together while points which do not have enough close neighbors are classified as outliers. Thus, we configure DBSCAN to output a binary decision (whether an instance is "fast" or "slow"). We also only mark at most one outlier at a time to make sure we do not remove too many service instances (explained later in §2.3).

Finally, we emphasize that we only compare instances (scores) of the same service. We do not compare instance scores of Cassandra with those of ZooKeeper, thus the algorithm above runs for every service deployed. For example in Figure 1, the ScoreDB server maintains history of Cassandra and ZooKeeper peer scores separately.

2.3 Mitigation

After a service instance is marked as an outlier, IASO starts the mitigation process. Below are the three options that our customers can set in IASO configuration. The first one (service instance reboot) is the default configuration. The philosophy of our mitigation is that it is better to remove a highly degraded instance than allowing it to induce a cascading problem to the entire cluster. Other works [24, 56] already show how running with one less instance ($N-1$) can give a better performance than running a full cluster (N) with a degraded instance. IASO only quarantines at most one instance to prevent the cluster drops below its fault-tolerant level.

(1) SERVICE INSTANCE REBOOT AND LEADERSHIP REMOVAL: Here, IASO will restart the slow instance and remove leadership leases (if any) from the service instance running on that node. We emphasize here that we only remove the service instance (*e.g.*, Cassandra/ZooKeeper slow instance), but not the underlying VM or the machine. As a reason, imagine a machine where an instance of service X uses the underlying slow disk, but another instance of service Y only utilizes the memory (still fast). Here, we want X to be rebooted and its leadership removed, but let Y continue to run normally as it is not affected by the slow disk.

Regarding the removal of leadership, in ZooKeeper, if the instance is a leader, rebooting the instance will automatically make ZooKeeper choose a new leader. This way the old slow leader is no longer the single point of performance failure. The only cost associated with this action is the rebuilding of leader state on some other healthy peer.

In Cassandra, every instance is responsible for a key range (our deployment does not use Cassandra's virtual node feature). Here, we have two opposing options for mitigation. The expensive option is to remove the instance from the ring and trigger a whole-cluster key-range rebalancing, which might be a premature action as the instance perhaps can be fixed soon. The cheaper option is to let the slow instance be in the ring but not allow it to be part of the data transfer.

We chose the latter option and modified Cassandra slightly to achieve this. In this mode, the slow instance is no longer the primary owner of its key range, but rather one of the other replicas becomes the primary owner. The upside is that we postpone the need for whole-cluster key-range rebalancing. The downside is that the fault tolerance of newly added data will be down by one (*e.g.*, we can only write to two replica nodes as the instance on the slow node is being isolated) and read throughput may be degraded due to the loss of one instance. We note that the fault tolerance of old data does not go down as the data is still there in the slow instance.

Regardless of the limitations of this default option, customers who have smaller clusters tend to choose this option as they do not have options to migrate the instance or VMs to another healthy machine. Below we discuss other options for customers with larger clusters.

(2) VM SHUTDOWN: This is a more severe action than the default option above. In this mode, the controller VM of the slow service instance is shut down and no services are started on the VM. The difference between this action and the default one above is that when VM is shut down, the services above will automatically run their recovery protocols (*e.g.*, whole ring rebalancing). Thus, the fault tolerance of the data stays the same (*e.g.*, 3-way replication is still maintained). The similarity is that there may also be a performance drop to the loss of a VM. When the problem is fixed, the VM is added backed and full performance can be restored.

(3) HOST MACHINE SHUTDOWN: This option is similar to VM shutdown. The difference is that our system will automatically migrate the entire VM from this host to another, which is a process transparent to the services running on the VM. There may be a potential VM rebalancing issue (*e.g.*, a machine has too many VMs). For VM balancing, we employ our own proprietary VM rebalancing that is outside the scope of the paper. We also emphasize that in our deployment, these machines are running the services that we deploy. The machines are not shared with other tenants, hence we have a full control of when to shut down the machine.

2.4 Resolution

The last stage, resolution, is the manual part of the whole IASO operational procedure, which we describe here for completeness.

When detecting a fail-slow node, IASO generates a user alert on the customer monitoring UI. IASO also pages our site-reliability engineers (SREs) such that they can work with the affected customer to fix the problem. If there had been a cluster outage (*i.e.*, cluster IOPS went to almost zero) before the mitigation, IASO helps the customer and our SREs in identifying the faulty node and service.

It is also possible that before the SREs perform the full

Components	LOC
Cassandra modification	585
ZooKeeper modification	199
IASO node-level library	547
ScoreDB server	3377

Table 1: **Implementation complexity (§2.5).** *The table shows our* IASO *integration effort.*

diagnosis, the problem already went away by itself from rebooting the slow node. We see this happens in cases such as CPU locks-ups or high heap usage levels. In such cases, IASO will no longer mark the node as a degraded node. In overall, when the problem is fixed, IASO immediately rolls back the fail-slow node actions executed before, and service instances on the newly recovered node regain their leadership responsibilities.

Temporary fail-slow faults can be recurrent (*e.g.*, high heap usage level). To prevent such recurrent faults, the root cause must be fixed. For example, we could apply some custom optimizations to our services to prevent it from entering such a state again.

2.5 Other Implementation Details

INTEGRATION: So far we have integrated IASO with Cassandra and ZooKeeper. The implementation complexity is shown in Table 1. The changes to the target services are non-intrusive (less than 600 LOC). The service instances use IASO library to measure local scores and send them to the ScoreDB server where the rest of the complexity lies. The total score data size of ScoreDB server is only 0.27 MB per day per cluster on average as it only needs to keep the score history of the last 10 minutes. The CPU overhead is near 0%.

We envision that IASO can be easily integrated to other master-worker systems where data flows across workers. For example, in HDFS, write replication forms a pipeline of datanodes where each datanode can sense the performance of its peers. For systems like ZooKeeper, the integration involves a different type of modification due to ZooKeeper's "pure" leader-follower architecture (*i.e.*, followers do not interact with each other). We describe these changes later below. As mentioned before, we also run our own blob-store service which can be integrated with IASO as well. This process is still in progress, not because of integration difficulty, but because so far our IASO integration in Cassandra and Zookeper seems to be sufficient. One limitation of our deployment is that a single blob-store instance can be misconfigured causing a fail-slow fault, but goes undetected (which again so far never happened).

ZOOKEEPER MODIFICATION: In our deployment, the Cassandra-side IASO so far has been very effective. But as we deal with deployments of tens of thousands of nodes, we can potentially cover a wider set of failure types if we can integrate IASO with another service as well. Hence, we attempted to integrate IASO to ZooKeeper, but ZooKeeper employs a pure leader-follower architecture where followers do not transfer data with each other (*i.e.*, 3-way writes flow from the leader to three followers, unlike in HDFS or Cassandra). The leader is a single point of performance failure [24]; if the leader's NIC is slow, the writes to all the followers will slow down, hence no outlier.

For this, we add a simple, lightweight background ping-pong thread between ZooKeeper peers (only <200 LOC). Every 10 seconds, every instance picks a maximum of 7 random peers and makes an RPC that includes a synchronous disk write. Checking the disk latency this way is also beneficial since most data operations in Cassandra hit the cache, hence disk monitoring is a bit lacking. Besides these small changes, we emphasize that the rest of the algorithm is the same – the instances send the median latencies of their peers (median of 1 minute window) to ScoreDB and the DBSCAN algorithm will compute the outlier.

THRESHOLDS: We would like to emphasize that the threshold values we use in our algorithms (§2.2) are based on our specific deployment experiences. It is possible that the values might not work in other cases.

3 Results

This section presents our experimental results, starting with unsuccessful experiences (§3.1) and then the successful ones (§3.2) and the false positive rates (§3.3).

3.1 Unsucessful Attempts

The first strawman approach we tried was to use the raw timeout count as a metric to sense service instance level performance degradation. Figure 2 shows the number of timeouts observed in three samples of real degraded instances (in different time periods and clusters). As shown, the timeouts observed occur in bursts although the fault is severe throughout the time interval. Thus, without saving the ratio of timeouts and responses for every peer over a given period, there is no way to detect whether these high scores were merely transient or if they were truly persistent and possibly catastrophic faults.

For this reason, we next attempted to create a more stable algorithm by defining a score to be the percentage of timeouts over the total responses in every epoch. The first line below is the same as the first equation in §2.2, and in the second line, a peer score is essentially the `ToRespRatio`.

```
ToRespRatio = #timeouts / #responses
score = ToRespRatio
```

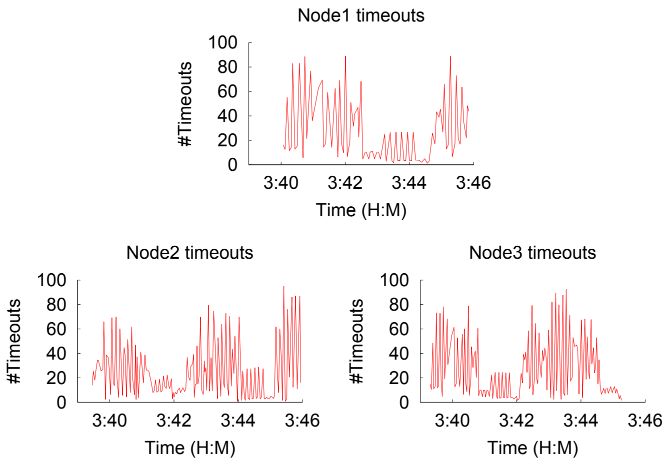


Figure 2: **Timeout fluctuations** (§3.1). *The figures show the number of timeouts observed over time in three samples of degraded nodes (different time period).*

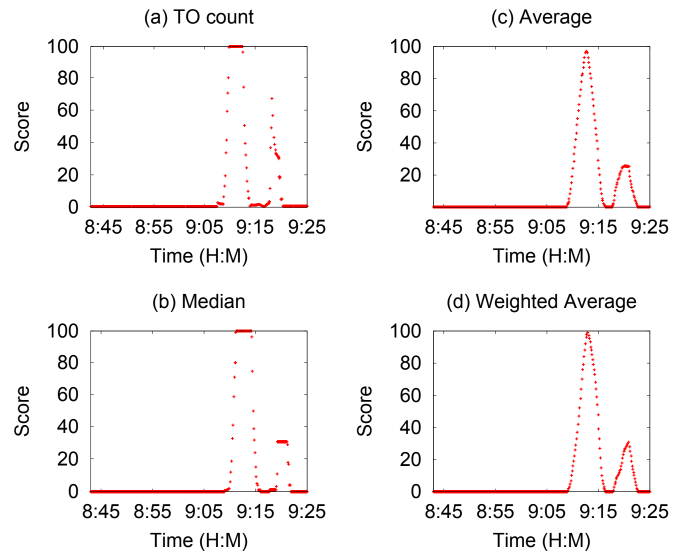


Figure 3: **Unstable scores** (§3.1). *Other attempts to create stable scores using timeout-response ratio as explained in §3.1.*

Figure 3a shows the result. Ideally the score should stay high throughout the degraded period, but instead we see one big spike and one small spike. We then modified the scoring algorithm slightly by using the median of the last 3-minute window:

$$\texttt{score} = \texttt{median}\left(\texttt{ToRespRatio in last 3 mins}\right)$$

The result, as shown in Figure 3b, still shows the same behavior (a dip between the two spikes). We tried replacing the median using average and weighted average and the result is similar (Figures 3c-d).

3.2 Successful Results

The previous section provides the reason we invented our custom outlier detection. Figure 4a shows the resulting scores from our custom algorithm, as detailed in Section 2.2. We can see that the metadata service (Cassandra/MS) instance on the degraded node has high scores assigned to it from 11:30 to 13:15 hours. Note that this is the case where we have not enabled the mitigation procedure, *i.e.*, the customer was experiencing degradation for almost 2 hours!

Correspondingly, to check that the scores are accurate, we checked the standard network performance graphs and we found that there had been a network issue at the exact time interval. Figure 4b shows the TCP SEND_Q size on the network connection between another node with this unhealthy node. Furthermore, Figure 4c shows the ping latencies to the degraded machine.

From these graphs, we can see that bad network performance on the slow machine correlated perfectly with the bad scores assigned to the nodes running on it. As a side note, we can see that the two metrics in Figures 4b-c cannot be used as raw scores as they also fluctuate.

Next, Figure 5 shows what is happening in the ScoreDB server side for a different fail-slow incident. The picture shows the representative scores by instance X measured for its $N-1$ peers on other nodes. For simplicity, the data here is from a cluster of 4 nodes. Figure 5a shows that `Node3` has a high score compared to other peers. But at this point `Node3` has not been marked as a definite outlier because its 30^{th}percentile score is not high yet. However, two minutes later, as shown in Figure 5b, we have sufficient scores for the 30^{th}percentile score to be high. When we plug this score into the DBSCAN algorithm, `Node3` was marked as a definite outlier.

IASO automatically quarantines an outlier to prevent it slowing down the entire cluster. Figure 6 shows another case after we deploy IASO. Here, the figure shows that the cluster-level IOPS drops to almost zero with the presence of one degraded machine, essentially showing how a degraded node can impact the entire cluster, as also shown by other works [24, 56]. Packet losses and the cluster-level degradation started occurring at around 09:15am but just after 10 minutes, IASO's mitigatory actions kicked in and the performance of the cluster was completely restored. Thus, with IASO, the time taken to quarantine a degraded node has now been brought down to the order of *minutes*. Note that the IOPS returns to "normal" although we lost a node, which is because in this scenario the 100K IOPS were far from the maximum throughput of the cluster.

3.3 True and False Positives

Figures 7a and 7b show the number of true and false positives we encountered every month across the 7 months, re-

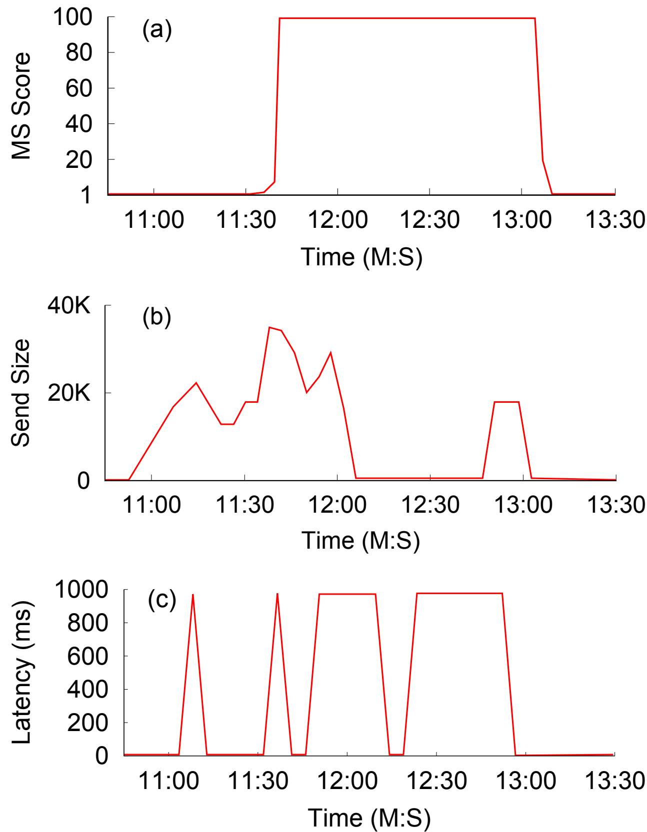


Figure 4: **Stable scores** (§3.2). *The figures show (a) the score of a degraded peer over time, (b) the SEND_Q size of the network connection to the degraded node, and (c) ping latencies to the degraded node monitored by our* `systat` *collector.*

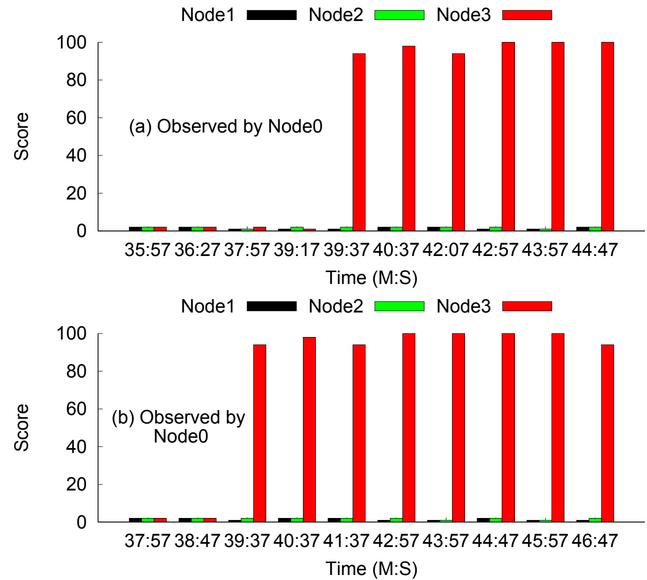


Figure 5: **Mitigation** (§3.2). *The top figure shows that* `Node3`*'s score is high as observed by* `Node0` *however it is not being marked as an outlier yet as its* 30th *percentile score is still low. In the bottom figure,* `Node3` *is marked as a definite outlier.*

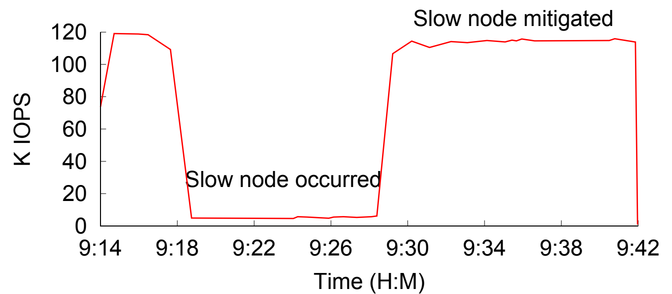


Figure 6: **Restored performance** (§3.2). *Within 10 minutes,* IASO *made the cluster-level IOPS return back to normal after isolating the slow node.*

spectively. For Figure 7b, the figure combines the number of "confirmed" and "probable" false positives as explained below.

Over a 7-month period, we encountered 9 *confirmed* false positives over the 232 true positives (confirmed fail-slow incidents), which brings our false positive rate to 3.7%. One major reason for our false positive is in our earlier versions of IASO where the cluster still sends data to a dead service instance and a healthy instance already becomes affected and "looks" slow as well. Here IASO incorrectly marks the healthy instance as an outlier. Due to space constraints, we put more false positive stories in our anonymized supplemental material [7].

We also encountered 41 *probable* false positives. We label these cases as "probable" because they do *not* necessarily suggest that IASO is imprecise. In these cases, by the time our SREs started debugging, the issue was no longer present and the service instances, VMs, and machines were healthy. Existing works gave some hints on the reasons behind this, for example, fail-slow incidents can be triggered by temporary environmental causes such as high temperature [28].

While we managed to record the false positives, we were not able to collect many false-negative reports (*i.e.*, undetected fail-slow incidents). This is because the reality of a large company and our SREs have their own priorities and might not contact us when they found cases that were not but should have been detected by IASO. The false negatives we were aware of came from two 2 outages that happened after the deployment of IASO, which can be found in our supplemental material [7]. Other false negatives we noticed include low workloads as fail-slow faults with low workloads might not necessarily result in timeouts. We did not fix this problem as almost all our customers heavily utilize their clusters.

From our perspective, we prefer false positives over false negatives as in our system IASO pages site-reliability engineers whenever it detects a fail-slow failure. This gives

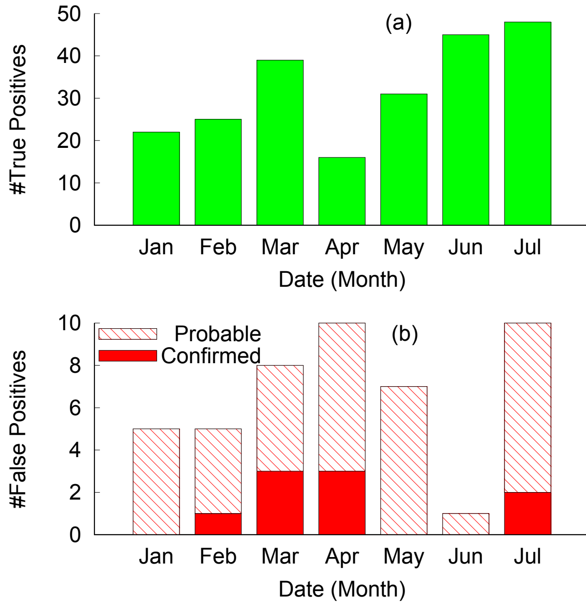


Figure 7: **True and false (confirmed+probable) positives (§3.3).** *The figure shows the number of (a) true and (b) false positives every month. The false positives include the "confirmed" and "probable" false positives as described in §3.3.*

us a way to easily track and investigate such issues and improve our system over time. As for the worst case impact, a false positive can cause a cluster to temporarily operate in a reduced fault tolerance state as IASO's extreme mitigation strategy can bring down a node. However, in case of a false negative, there can be an entire cluster outage which can stay undetected for hours.

4 Fail-Slow Dataset and Analysis

The deployment of IASO allows us to analyze fail-slow incidents in our vast field of clusters, which then enables us to perform new statistical studies. This section first describes our dataset (§4.1) followed with our findings (§4.2).

4.1 Dataset

We first describe our deployment settings. Our field consists of 39,000 nodes spread across many clusters. A cluster size ranges from 3 to 56 nodes. Our various cluster models and configurations (RAM size, storage, etc.) can be found in our supplemental material [7]. A cluster can contain heterogeneous nodes as we support heterogeneous applications and a broken hardware can be replaced with a higher-end one. Each node in a cluster runs a special VM called a controller VM where our data and control path services run. Among these services, Cassandra and Zookeeper run with IASO integration.

Failure	AFR	Notes
SSD error	5-15.7%	$\geq$ one uncorrectable error [53]
SSD failure	1-2%	Dead SSDs [16]
Disk error	1.7-8.6%	$\geq$ one failure event [46, 52]
DRAM error	2.2-9.0%	$\geq$ one memory error [33, 54]
fail-slow	**1.02%**	Node-level fail-slow faults

Table 2: **Fail-slow AFR (§4.2.1).** *Comparisons of annual failure rates of different types of failures*

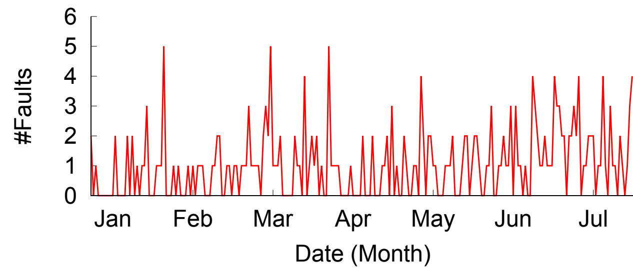


Figure 8: **Fail-slow per day (§4.2.1).** *The figure shows the number of fail-slow incidents per day in our field over 7 months.*

As mentioned before, every time IASO detects a fail-slow fault, it raises an alert that triggers the opening of a *support ticket* to investigate the issue. The support case is investigated by a team of trained site reliability engineers (SREs), who in turn coordinate with the customer and debug the issue. Once the problem is identified, the SREs update the support ticket with a category of the root cause found and the steps to resolve the issue. Other information that is updated as part of the case includes the time of the incident, a cluster identification number, the software version on the customer's cluster, the model family of the node that was affected and the number of months the node has been with the customer at the time of the incident.

With 232 fail-slow related tickets, our dataset can be seen as the largest fail-slow data from within a company. The previous largest dataset was 101 cases from 12 different institutions (more in §5). The next section presents our findings from studying the support tickets. The dataset that we will make public and discuss here comes from a period of seven months in 2017. The dataset for 2018 is still being perused and cleaned, hence not part of this submission.

4.2 Findings

4.2.1 Frequency

With a large dataset, we are able to measure the annual failure rate (AFR) of fail-slow incidents. Given 232 independent cases across 39,000 nodes over 7 months, we can derive that fail-slow AFR is 1.02% (232 $\times$ 12 / 7 / 39,000).

Table 2 compares fail-slow AFR with the rates of other types of failures. As shown, fail-slow fault frequency is rel-

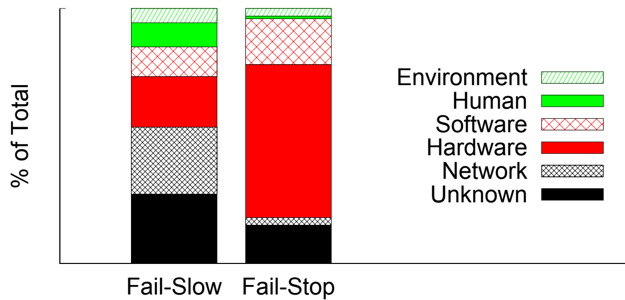


Figure 9: **Fail-slow root causes** (§4.2.2). *The figure shows the breakdown of fail-slow root causes (and the comparison to fail-stop causes).*

atively significant and cannot be ignored. Figure 8 breaks down the number of fail-slow incidents observed per day in our field over the 7 months. We see that barring a couple of days in between, there is at least one failure per day. These statistics accentuate the importance of fail-slow detection and mitigation frameworks such as IASO.

4.2.2 Root Causes

Next, we analyze the root causes of fail-slow incidents. To compare the frequencies of various different root causes of fail-slow incidents with those of fail-stop failures [51], we group the causes into six categories: `Hardware`, `Software`, `Network`, `Environment`, `Human` and `Unknown`. For example, all issues that had a tag of "memory" or "disk" in our support tickets are grouped under `Hardware`.

Figure 9 shows the breakdown of fail-slow root causes (and the comparison to fail-stop causes from a related work [51, Figure 4a]). `Hardware` and `Network` failures turn out to be the highest contributors of fail-slow incidents in our field. Their total is roughly the same as in the fail-stop cases. In the next section, we break down the *sub*-causes to understand more about the root causes.

The `Unknown` count is quite significant because of a couple of reasons. One common reason is when a customer becomes unresponsive during the support case or does not want the issue to be investigated further without providing a clear reason. We believe this can be either because the customer did not notice any issue around the time the fail-slow alert was generated (thereby a false positive) or fixed the issue themselves without our help. The other reason is when the SREs could not find a specific root cause for the issue or did not tag the support case with a clear cause.

4.2.3 Root Sub-Causes

Table 3 shows further the breakdown of the sub-causes within each of the five root categories in the previous section. The numbers in the parentheses are the count of tickets.

Root	Sub-causes
Hardware	Faulty dimm (15), ECC error (10), low memory (9), SATADOM (5), CRC error (1), RAID controller (1), LSI controller (1), unknown (5)
Software	Software upgrade (8), VM issue (6), GC (3), BIOS (1), scheduler (1), unknown (6)
Network	Faulty device (13), network outage (9), device replace(7), unreachability (6), packet drop (5), network contention (2), device reboot (1), unknown (18)
Environment	Incorrect setting (11), high load (1), energy issue (1)
Human error	Misconf (10), network migration (4), install /deploy (3), unplugged cable (2), unknown (4)

Table 3: **Root *sub*-causes** (§4.2.3). *The table shows the sub-causes within each of the five categories of known root causes. The dataset will be released publicly.*

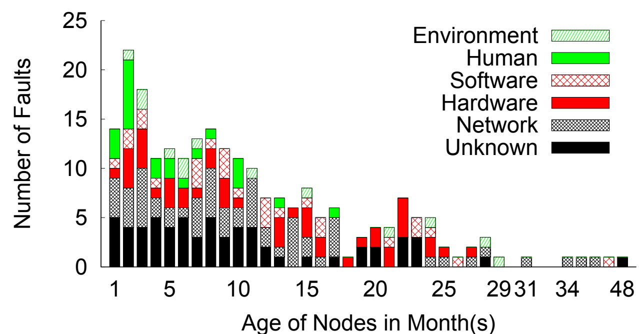


Figure 10: **Fail-slow vs. age** (§4.2.4). *The figure correlates fail-slow incidents with machine ages.*

For example, for hardware-induced slowdown, it can be because of faulty dimm, ECC/CRC errors, low memory, etc., while network-induced slowdown can be because of faulty NICs/switches, bad cables, packet drops, and network contention.

Our goal here is to show that fail-slow root causes *vary widely*. We believe this is a strong motivation why fail-slow detection and mitigation should be also deployed at the service level (not just low-level hardware level). Our findings are also consistent with those reported in a recent paper [28]; we observed in our field how fault conversions take place and how different failure types such as fail-stop (*e.g.*, disk/SSD failure), fail-transient (*e.g.*, GC), and fail-partial (ECC errors) can transform into fail-slow failures at the service level [28, §3.2].

4.2.4 Age and Model

As our ticketing system automatically collects machine age data, we are able to correlate fail-slow failures with machine ages, as shown in Figure 10, bucketed into months ranging

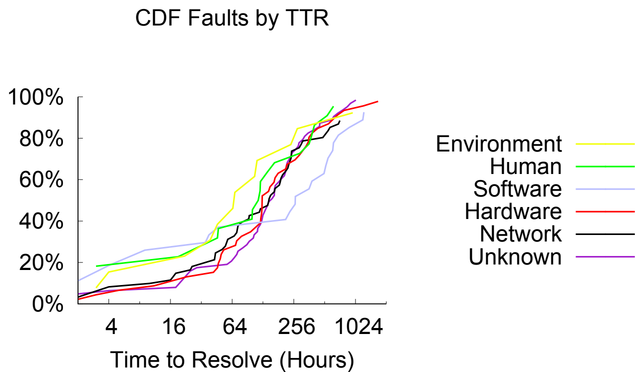


Figure 11: **Tickets TTR (§4.2.5).** *The figure shows the CDF of time to resolve tickets across different root-cause categories.*

	Net	Unk	HW	SW	Human	Env
Median	79	145	126	234	108	65
Mean	149	220	244	323	165	149
Max	721	1033	1705	1238	625	964

Table 4: **TTR tickets (§4.2.5).** *The table shows the median, mean, and maximum values of the data in Figure 11.*

from 1 to 48. We can see the "infant mortality" trend where younger machines exhibit more issues, but older (perhaps more stable) machines exhibit fewer issues. This follows the same failure trend in fail-stop failures [51, Figure 4]. This also supports a continuous paradigm where when the rate of fail-stop errors drops so does the fail-slow ones.

We also attempted to correlate fail-slow failures with the node model family and found no significant correlation, that every node model family suffers from faults across a majority of component types (see [7] for more).

4.2.5 Tickets TTR

Finally, Figure 11 shows the distribution of time to resolve the tickets (in hours) across different root causes. Table 4 shows the median, mean, and max values of the data in Figure 11. We emphasize that this metric does *not* represent the time for IASO to mitigate the issues (which is in the order of 10 minutes), but rather how long it takes to close a ticket. When a ticket is closed, the customer's cluster is guaranteed to be back fully healthy.

The reason we show this data is to point out that a fail-slow root cause can take *days* to be fully resolved. This is consistent with anecdotal experiences shared by large-scale operators from various institutions [28, §3.5]). Hence, it is important to quickly quarantine the fail-slow component before the performance problem cascades to the entire cluster.

	HW	SW	Service
Bug finding	SymDrive[47], DDT[39]	MacePC[38], PCatch[40], SPV[55]	Orca[15]
Detection	IPMI[2], SNMP[3], SMART[4], Ganglia[42]	UBL[20], Toddler[43]	PeerReview[30], AFD[45]
Diagnosis	Roy[49], PerfBlower[25]	Xray[13], Hytrace[19], PerfScope[21], PerfCompass[22], Deepview[59], Stitch[60], FaultLocalize[50]	Canopy[36], PivotTracing[41], Pip[48], Panorama[31]
Mitigation	Carburizer[35], DisturbMLC[14], VibrateSSD[17]	Mantri[11], DeepDive[44], PBSE[56]	PREPARE[57], **IASO**

Table 5: **Related work (IASO).** *The table categorizes works that relate to fail-slow detection, diagnosis, and mitigation across hardware-, software-, and service levels.*

5 Related Work

We now discuss related work beyond the papers that we already cited earlier. In particular, we break the discussion here to two categories: (1) works related to fail-slow detection and mitigation systems and (2) publications that release information about fail-slow incidents.

Table 5 shows that there are many tools, frameworks, and approaches that have been introduced or deployed for different levels of the hardware, software, and service stack. First, there are many *bug-finding* tools such as MacePC [38], PCatch [40], and Orca [15], but they are offline approaches. Second, there are online fail-slow *detection* tools across the hardware/software stack. For example, SMART [4] is a monitoring tool that can be used to detect hardware degradation but does not include diagnosis capability. Third, Pip [48], PivotTracing [41] and many others provide *diagnosis* approaches that work at the service level (not just one particular software) but they do not make quarantine decisions. Finally, IASO is in a category that performs detection and automated *mitigation*. In this space, we are not aware of many published works. The limitation of IASO is that it does not come with diagnosis tools. Thus, the diagnosis approaches in the 3rd row of Table 5 are orthogonal to our work.

Table 6 shows publications that release datasets on performance problems. The table shows the year span (Yr), number of fail-slow failures/bugs reported ($\#F$), deployment size/number of nodes ($\#N$), the number of systems/services the data is collected from ($\#S$) and the scope of the root-cause analysis (A). The top part of the table represents incidents that appear in live deployments while the bottom of the

Related Work	Yr	#F	#N	#S	A
IASO	'16-17	232	39k	1	ehmnsu
Fail-slow[28]	'00-17	101	≥10k	12	hn
GrayFailure[32]	-	4	-	1	-
Panorama[31]	'17-18	15	20	4	-
COS[27]	'09-15	126	-	32	ehmnsu
CBS[26]	'11-14	860	-	6	s
PerfBugs[34]	'00-11	109	-	5	s
Limplock[24]	'13	28	≥30	5	s

Table 6: **Related work (fail-slow dataset).** *For each related work, the columns show the year span (Yr), number of fail-slow failures/bugs reported ($\#F$), deployment size/number of nodes ($\#N$), the number of systems/services the data is collected from ($\#S$) and the scope of the root-cause analysis (A). In the last column (analysis), "h" represents hardware, "s" software, "n" network, "e" environment, and "m" human. Papers with "s"-only label implies bug-study papers.*

paper represents works that study/test software bugs. In the former category, our dataset can be considered as the largest dataset of fail-slow cases publicly reported from within a company. Our work strongly supplements existing anecdotes that fail-slow faults at all levels, hardware and software, have to be addressed.

6 Conclusion

We have described our successful 1.5-year deployment of IASO. We found fail-slow detection and automated mitigation schemes are crucial in preventing fail-slow induced outages in our large deployment field. We would like to emphasize again that automatic fail-slow mitigation/quarantine schemes (beyond detection only) are relatively a new area of research. We hope our paper can provide insights to the development of better frameworks in the future.

As future work, we look forward to building a more aggressive algorithm that can quarantine a slow node shorter than our current 10-minute interval (and do so with low false positives) as well as automatically marking fail-slow faults that are resolved by themselves without depending on our customers or SREs (more in [7]). Furthermore, as we continue to collect peer scores reported in the field, we hope to learn more detailed characteristics.

7 Acknowledgments

We thank Ric Wheeler, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Roger Liao and Anshul Purohit for their significant contributions during the development of IASO. We learnt a lot from the actual customer cases where IASO was effective and also from a few scenarios where we hit false positives. Thanks to Rob Savino and Mark Czarnecki for their effort to make this information available to us and help us with quite some root cause analysis. University of Chicago authors were supported by funding from NSF grant No. CNS-1350499.

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Intelligent Platform Management Interface (IPMI). https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html.

[3] Simple Network Management Protocol (SNMP). http://www.net-snmp.org/.

[4] S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology). http://en.wikipedia.org/wiki/S.M.A.R.T.

[5] Apache Cassandra NoSQL Performance Benchmarks. https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks, 2018.

[6] Density-based spatial clustering of applications with noise. https://en.wikipedia.org/wiki/DBSCAN, 2018.

[7] Iaso Supplementary Materials (Anonymized). https://tinyurl.com/iaso-supplementalmaterial, 2018.

[8] Iaso Wiki. https://en.wikipedia.org/wiki/Iaso, 2018.

[9] Hyper-converged infrastructure. https://en.wikipedia.org/wiki/Hyper-converged_infrastructure, 2019.

[10] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[11] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *Hot Topics in Operating Systems*, 2001.

[13] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[14] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2015.

[15] R. H. Campbell and S. M. Tan. μChoices: An Object-Oriented Multimedia Operating System. In *In Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[16] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. Charaterizing private clouds: A large-scale empirical analysis of enterprise clusters. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[17] Christine S. Chan, Boxiang Pan, Kenny Gross, Kenny Gross, and Tajana Simunic Rosing. Correcting vibration-induced performance degradation in enterprise servers. In *The Greenmetrics workshop (Greenmetrics)*, 2013.

[18] Chiu, Dah-Ming, and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 1989.

[19] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[20] Daniel J. Dean, Hiep Nguyen, and Xiaohui Gu. UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems. In *Proceedings of the 9th ACM International Conference on Autonomic Computing (ICAC)*, 2012.

[21] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[22] Daniel J. Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(6), June 2016.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[24] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[25] Lu Fang, Liang Dou, and Guoqing Xu. PERFBLOWER : Quickly Detecting Memory-Related Performance Problems via Amplification. In *29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[26] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[27] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim

Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[29] Ashish Gupta and Jeff Shute. High-availability at massive scale: Building google's data infrastructure for ads. *Proc. of BIRTE*, 2015.

[30] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[31] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[32] Peng Huang, Chuanxiong Guo, Lindong Znhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randonph Yao. Gray Failure: The Achilles' Heel of Cloud Scale Systems. In *The 16th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2017.

[33] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[34] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[35] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[36] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, 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. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[37] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[38] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[39] Volodymyr Kuznetsov, Vitaly Chipounov, George Candea, École Polytechnique Fédérale de Lausanne, and Switzerland. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[40] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Dongsheng Li, and Xicheng Lu. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[41] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[42] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[43] Adrian Nistor, Linhai Song, Darko Marinov, and Shan L. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[44] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive:Transparently Identifyingand Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[45] Husanbir S. Pannu, Jianguo Liu, Qiang Guan, and Song Fu. AFD: Adaptive failure detection system for cloud computing infrastructures. In *31th IEEE – International Performance Computing and Communications Conference (IPCCC)*, 2012.

[46] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[47] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing Drivers without Devices. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[48] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive Realtime Datacenter Fault Detection and Localization. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[50] Swarup Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[51] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[52] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[53] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[54] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[55] Riza O. Suminto, Agung Laksono, Anang D. Satria, Thanh Do, and Haryadi S. Gunawi. Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems. In *The 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2015.

[56] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[57] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[58] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[59] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[60] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

PARTISAN: Scaling the Distributed Actor Runtime

Christopher S. Meiklejohn, Heather Miller
Carnegie Mellon University

Peter Alvaro
UC Santa Cruz

Abstract

We present the design of an alternative runtime system for improved scalability and reduced latency in actor applications called PARTISAN. PARTISAN provides higher scalability by allowing the application developer to specify the network overlay used at runtime without changing application semantics, thereby specializing the network communication patterns to the application. PARTISAN reduces message latency through a combination of three predominately automatic optimizations: *parallelism*, *named channels*, and *affinitized* scheduling. We implement a prototype of PARTISAN in Erlang and demonstrate that PARTISAN achieves up to an order of magnitude increase in the number of nodes the system can scale to through runtime overlay selection, up to a 38.07x increase in throughput, and up to a 13.5x reduction in latency over Distributed Erlang.

1 Introduction

Building distributed applications remains a difficult task for application developers today due to the challenges of concurrency, state management, and parallelism. One promising approach to building these types of applications is by using distributed actors; the actor-based programming paradigm is one where actors can live on different nodes and communicate transparently to actors running on other nodes. Actor-based programming is well suited to the challenges of distributed systems; actors encapsulate state, allowing controlled, serial access for state manipulation. A single machine can typically run hundreds of thousands of actors, allowing efficient use of resources per machine and thereby enabling high-scalability and high-concurrency by elastically scaling the number of machines in a cluster. Taken together with the fact that actors communicate through unidirectional asynchronous message passing with no shared memory between them, the actor-based programming paradigm is well suited to the nature of distributed systems. In addition to providing developers of distributed systems with a convenient programming model, distributed actor systems can also be efficiently implemented, which has resulted in significant adoption and large-scale success in many areas of industry.

There exist three primary industrial-grade distributed actor systems; Distributed Erlang [31], Akka [21] (for Scala) and Microsoft's Orleans [8, 10] (for C#). Distributed Erlang has been used as the underlying infrastructure for message brokers [2, 25], distributed databases [4, 6, 18], and has provided infrastructure for the chat functionality for applications like *WhatsApp*, *Call of Duty*, and *League of Legends*. [14, 15, 27] Similarly, Akka has been used by Netflix for the management of time series data [23], and Microsoft's Orleans has been used as the underlying infrastructure for Microsoft's popular online multiplayer games, *Halo* and *Gears of War* for the Xbox [24]. In all of these cases, these applications have benefited from both the state encapsulation and pervasive concurrency that actors provide and the fault isolation of actors by reducing the use of shared memory. However, these distributed actor systems are still limited in terms of both *scalability* and *latency*.

Scalability. Compared to other distributed frameworks which can support hundreds to thousands of nodes, these production-grade distributed actor systems are still limited in the number of nodes that they can support. Distributed Erlang, for instance, has not been operated on clusters larger than 200 nodes [1], whereas one of the more popular applications built on Distributed Erlang, the distributed database Riak, has been demonstrated to not scale beyond 60 nodes [15]. As we will later show, this limited scalability is related to the rigidity of the overlay network—the communication pattern between the nodes in the application—used in the runtime system. This rigidity has been the subject of previous research on alternative designs to improve the scalability of the system [11], and efforts to find a "one-size-fits-all" overlay, which can equally serve all types of distributed applications, have not been successful [28]. Thus, especially in the context of Distributed Erlang, scalability is still a major challenge.

Latency. Due to their underlying model of computation—unidirectional asynchronous message passing between ac-

tors with independent queues that are multiplexed onto a single queue between nodes—distributed actor systems frequently suffer from the problem of head-of-line blocking. For example, the distributed database Riak avoids using Distributed Erlang for background data synchronization (e.g., hinted and ownership handoff) to avoid head-of-line blocking in the read/write request path. While alleviating head-of-line blocking has been the subject of much research [12, 30] and remains a relevant problem in today's large-scale systems [9], the general solution of introducing more queues and partitioning communication across those queues does not necessarily yield better performance without a priori knowledge of the application's workload.

Application-specific information exists that can be used to reduce the effects of head-of-line blocking. Given (i) the knowledge of the identities of the actors that are sending messages, (ii) the identities of the recipients, and (iii) the knowledge that actors will process their messages sequentially, this application-specific information can be provided in the form of a small number of lightweight annotations to the runtime. These annotations can help the runtime to separate network traffic over specialized channels (e.g., cluster maintenance, high-priority application behavior, failure detection), in turn leading to the reduction of head-of-line blocking in an application-specific manner.

In this paper, we present the design of an alternative runtime system for improving the scalability and performance of distributed actor systems, along with an implementation of this runtime called PARTISAN. PARTISAN enables greater scalability by allowing the application developer to specialize the overlay network to the communication pattern required by the application at runtime without altering application semantics. PARTISAN facilities lower latency by providing the application developer with three ways to customize messaging behavior, without altering application semantics or requiring changes to application code. PARTISAN enables the application developer to (i) customize *parallelism* (for increasing the number of communication channels between nodes), (ii) utilize *named channels* (for separating different types of messages sent between actors), and (iii) *affinitize* scheduling (for partitioning traffic across communication channels depending on message source, destination and type).

We implement PARTISAN using Erlang without requiring changes to the Erlang VM, in an effort to make these scalability and latency benefits immediately available to production Erlang applications with minimal changes to application code. We provide a detailed experimental evaluation which, beyond microbenchmarks, includes a port of an existing widely-deployed Erlang distributed computing framework to take advantage of PARTISAN's optimizations. In our evaluation, we demonstrate that the use of each of these optimizations independently results in latency reduction, but the combination of these techniques yields significant reductions in latency.

The contributions of this paper are the following:

- We present the design of the PARTISAN runtime system that enables the runtime selection of overlay, enabling greater scalability by specializing the overlay to the application's communication patterns (Sections 3 & 5);
- We present a collection of predominantly automatic optimizations for latency reduction, realized in PARTISAN, that enable more efficient scheduling of messages on the network, specifically by exploiting (i) *parallelism*, (ii) *named channels*, and (iii) *affinitized* scheduling (Sections 4 & 5);
- We provide an open source implementation of PARTISAN that supports the runtime selection of overlay with implementations of four different overlay networks (Section 5);
- We port an existing widely-deployed open source distributed computing framework, Riak Core, from Distributed Erlang to PARTISAN, and provide an analysis of the process (Section 6);
- We present a detailed empirical evaluation of PARTISAN on (i) microbenchmarks, (ii) an industrial-grade actor-based distributed programming framework (Riak Core), and (iii) a research framework for distributed programming over replicated shared state (Lasp). We go on to show that PARTISAN demonstrates greater scalability (in some experiments, an order of magnitude increase in the number of nodes the system can scale to) through runtime overlay selection and lower latency (in some experiments, up to a 38.07x increase in throughput, and a 13.5x reduction in latency) through latency reduction optimizations (Section 6).

2 Background: Distributed Actors

Actors provide a simple programming model for building highly concurrent applications. Programming with actors involves two primary concepts: actors: lightweight processes that act sequentially, respond to messages from other actors, and sent messages to other actors; and asynchronous message passing: unidirectional, asynchronous messages that are sent between actors. Applications built using the actor model typically achieve their task through the cooperation of many actors sending messages to one another. No state is shared between actors: the only way for data to be shared between actors is through message passing[1]. Actors are designed to be extremely lightweight and typically implementations allow for ten to hundreds of thousands of actors per machine. As no data is shared, and actors are relatively independent with loose coupling to other actors – strictly through message passing – if a particular actor happens to fail, the fault remains isolated to that actor. Actors are not static: actors are allowed to "spawn" other actors as the system is running.

[1] Pony is a unique exception here, which uses a capability system to know when it is safe to share memory. However, this is an implementation detail as the programming model remains that of message passing.

Actors are a popular mechanism for building highly concurrent applications as they allow both users and user actions to be modeled as actors themselves. For instance, in the aforementioned *Halo* and *Call of Duty* examples, actors are use for modeling the presence service for the online functionality of the game. Therefore, a single actor, dynamically created, is used to model a connection to the service for a single user. In the Riak distributed database, an actor is spawned for every single read or write request made to the database. As the number of actors can range several orders of magnitude higher than the parallel computing capacity of a single machine, preemptive (e.g., Erlang) or cooperative scheduling (e.g., Orleans) is used for actor scheduling within the runtime.

Distributed actor systems extend the actor functionality from a single machine to a cluster of machines. Distribution adds a number of complexities to the model: (i) *failure detection*: actors may be unavailable under network partitions or crash failures of remote machines; (ii) *message omission*: messages are no longer guaranteed to arrive at a destination due to failure; (iii) *membership*: or what nodes are currently members of the cluster and how the membership overlay is organized; (iv) *binding*: the location of actors may not be known at runtime when actors are dynamically created; (v) *contention*: contention for access to network resources may slow down actors; (vi) *congestion*: and the varying location of actors results in non-uniform latency with inter-actor messaging when actors are located on different machines.

2.1 Framework Commonalities

These concerns are addressed by the contemporary industrial distributed actor systems through various mechanisms. Each of these mechanisms introduces additional network overhead that the application developer may not be aware of, contributing to reduced scalability and higher latencies.

Failure detection. Actors may become unreachable due to crash failures or network partitions. To detect failures, nodes typically send heartbeat messages to the other nodes in the cluster. When a node is suspected as failed, it's assumed that the actors that were running on that node failed.

Message omission. Distributed actor systems try to address the problem of message omission by using TCP. With a single connection, TCP ensures FIFO ordering of messages between pairs of actors and best-effort delivery using retransmission based on sequence numbers and acknowledgements.

However, as failure detection is imperfect and nodes may be disconnected and reconnected under network partitions or crash failures, message delivery is not guaranteed by the runtime system. Therefore, distributed actor systems typically require the user to program as if message omission is always a possibility. Put more generally, TCP connections are session-oriented and in these frameworks delivery guarantees do not hold across sessions.

Membership. Membership determines which nodes are part of the cluster and are available for hosting actors. Failure detection is combined with membership to determine who the active members of the cluster are at any given moment.

Binding. When sending a message from one actor to another, the location of that actor may or may not be known at a given time. Most of these systems encode a node identifier into the process identifier, or leverage a replicated, global process registry, for determining the location of an actor by a registered name instead of a process identifier.

2.2 Challenges

The problems of both network contention and network congestion remain challenges for distributed actor systems.

Network contention. All of the aforementioned actor systems support inter-machine communication through the use of a single TCP connection, therefore multiplexing actor-to-actor communication on a single channel. Not only does actor-to-actor communication (data) use this channel, but background communication from the membership and failure detection systems (control) also contribute to congestion on this link. Taken together with CPU-intensive activities that may block access to the socket (message serialization/deserialization, for example) and non-uniform distribution of message load (slow-senders vs. fast-senders), the possibility for contention increases, which in turn increases latency and reduces throughput of the system. This is further exacerbated by certain overlays; for example, the full-mesh overlay must perform failure detection from all nodes to all other nodes.

Network congestion. Network congestion, in the form of latency or congestion control, may further impact performance. Under situations where the frequency of message sends exceeds what can be transmitted over the network, causing queueing delays on these multiplexed connections between nodes, other senders on the same node may be penalized and forced to wait for other senders to transmit.

3 Overlay Networks

To address the problems that arise from a fixed overlay, PARTISAN supports the selection of overlay at runtime. PARTISAN's API exposes an overlay agnostic programming model – only asynchronous messaging and cluster membership operations – that easily allows programmers to build applications that can operate over any of the supported overlays. Selection of the overlay at runtime only affects the performance of the application, and does not change the application semantics. Selection of the overlay is done with a configuration parameter specified at runtime; therefore, changing the overlay does not require recompilation and the selection is fixed for the lifetime of the application.

PARTISAN supports four overlays and exposes an API for developers to extend the system with their own overlays: *static*, *full-mesh*, *client-server*, and *peer-to-peer*.

3.1 Static, Full-mesh, Client-server Overlays

The static, full-mesh, and client-server overlays are similar. Each overlay uses a single connection for communication between each node in the cluster. Failure detection is performed by monitoring this connection; when this connections is dropped, the node is reported as down.

With the static overlay, membership is fixed at runtime whereas with the full-mesh overlay, membership is dynamic and can be altered while the system is running. With the client-server overlay, connections are only maintained between servers and from servers to clients, similar to a traditional hub-and-spoke topology.

3.2 Peer-to-peer Overlay

The peer-to-peer overlay builds upon the HyParView [20] membership protocol and the Plumtree [19] broadcast protocol, both of which use a two-phase approach to pair an efficient dissemination protocol with a resilient repair protocol used to ensure operation during network partitions.

HyParView. HyParView is an algorithm that provides a resilient membership protocol by using partial views to provide global system connectivity in a scalable way. Using partial views ensures scalability; however, since each node only sees part of the system, it is possible that node failures break connectivity. To overcome this, HyParView uses two different partial views that are maintained with different strategies.

Plumtree. Plumtree is an algorithm that provides reliable broadcast by combining a deterministic tree-based broadcast protocol with a gossip protocol. The tree-based protocol constructs and uses a spanning tree to achieve efficient broadcast. However, it is not resilient to node failures. The gossip protocol is able to repair the tree when node failures occur.

Semantics. However, with partial views, nodes may want to message other nodes that are not directly connected. To maintain the existing semantics of existing actor systems, PARTISAN needs to support messaging between any two nodes in a cluster. To achieve this, PARTISAN's peer-to-peer membership backend uses an instance of the Plumtree protocol to compute a spanning tree rooted at each node. When sending to a node that is not directly connected, the spanning tree is used to forward the message down the leaves of the tree in a best-effort method for delivering the message to the desired node. This is similar to the approach taken by Cimbiosys [26] to prevent livelocks in their anti-entropy system.

4 Latency Reduction

In Section 2, we discussed a number of features of distributed actor systems that operate in the background to maintain cluster operation. These included *binding*, *membership*, and *failure detection*. Each of these features of actor systems can be expensive in terms of network traffic and contributes to increasing the overall message latency by delaying application-specific messaging behind cluster maintenance messaging. In addition to background traffic, it's also possible that one type of application-specific messaging may also delay different types of application-specific messaging, as in the case where a slow sender is arbitrarily delayed behind a fast sender. These are all specific cases of head-of-line blocking.

To alleviate these issues, we provide the application developer with three ways to customize messaging behavior in a distributed actor system; by (i) customizing *parallelism*, (ii) utilizing *named channels*, and (iii) *affinitized* scheduling.

4.1 Parallelism

To reduce the effects of head-of-line blocking with a single message queue, additional message queues can be introduced in an attempt to parallelize as much work as possible. We refer to this mechanism as *parallelism*. With little input from the application developer—only a specification of the number of queues to operate at each node for each destination node—the system can either use random or round-robin scheduling to assign work to queues. In most cases, the system can optimally choose this parameter based on available system resources.

4.2 Named Channels

While parallelism serves to increase the amount of work performed in parallel, background messages may be queued in front of application-specific messages, resulting in diminishing returns if this is the only technique used to reduce latency.

If we further classify these message queues as either queues for background messaging or application-specific messaging, we can be more intelligent in our scheduling. This can be achieved using *named channels*, and it is similar to Quality-of-Service (QoS) present in many modern networking systems. This mechanism only requires the application developer to annotate what type of message is being sent, and dedicated queues based on type are used for scheduling these messages. This mechanism allows the system to automatically place background messaging on a queue where it will not interfere with application-specific messaging.

4.3 Affinity

While named channels prevent background messaging from directly interfering with application-specific messaging, application-specific messaging may still suffer from interference between actors that send at different rates.

Under the assumption that multiple outgoing queues are available (parallelism), random or round-robin scheduling may still produce schedules that lead of head-of-line blocking issues. With the knowledge that actors have (i) a distinct identity (unique references which point to each actor and which can itself be exchanged), (ii) and act sequentially, we can further refine our message scheduling algorithm by selecting an outgoing message queue based on the sending actor's identity.

Feature	API	Analogous Call (Erlang)
Join node to cluster	`join(Node)`	`net_kernel:connect_node(Node)`
Remove self from the cluster	`leave()`	`net_kernel:stop()`
Return locally known peers	`members()`	`nodes()`
Forward message to registered name	`forward(Node, Name, Msg, Opts)`	`erlang:send({Name, Node}, Msg)`
Forward message to process id	`forward(Pid, Msg, Opts)`	`erlang:send(Pid, Msg)`

Table 1: PARTISAN's API

```
call(Dst, Msg, Timeout) ->
  Dst ! Msg,

  receive
    Response ->
      Response
  after
    Timeout ->
      {error, timeout}
    end
end.
```

(a) Distributed Erlang

```
call(Dst, Msg, Timeout) ->
  partisan_pluggable_peer_service_manager:forward(Dst, Msg, []),

  receive
    Response ->
      Response
  after
    Timeout ->
      {error, timeout}
    end
end.
```

(b) PARTISAN

Listing 1: Sending messages using Distributed Erlang and PARTISAN. PARTISAN's API is designed to be a drop-in replacement for Distributed Erlang.

```
%% Use `N' to partition with affinitized scheduling.
partisan_pluggable_peer_service_manager:forward(
    Dst, Msg, [ {partition_key, N} ])

%% Use `Channel' to partition by channel.
partisan_pluggable_peer_service_manager:forward(
    Dst, Msg, [ {channel, Channel} ])
```

Listing 2: Sending messages using PARTISAN. PARTISAN's API allows both affinitized scheduling and channels to be specified for a single message send.

This scheduling technique is known as *affinitized* scheduling and results in a further reduction in latency for network intensive processes by avoiding interference between different actors that send messages at different rates—for example, two actors on the same node sending at different rates to the same remote actor can be scheduled on different queues.

The application developer can take advantage of affinitized scheduling either by enabling affinitized scheduling for all messages, where a partition key is automatically derived by the system, or by annotating individual message sends with a partition key. This partition key is then concatenated with the identity of the recipient and, using a hash function, is used to select the appropriate queue. By hashing both the sender and the recipient together, the system will attempt to collocate pairwise communication between the same two actors together, providing best-effort FIFO when the system is not operating under failure.

5 PARTISAN

PARTISAN is a runtime system that enables greater scalability and reduced latency for distributed actor applications. PARTISAN improves scalability by allowing the application developer to specialize the overlay network to the application's communication patterns. PARTISAN achieves lower latency by leveraging several predominately automatic optimizations that result in the efficient scheduling of messages. PARTISAN is the first distributed actor system to expose this level of control to the application developer, improving the performance of existing actor application and enabling new types of actor applications.

5.1 Design

All three industrial-grade actor systems follow the same underlying assumptions that define the actor model. The design of PARTISAN is therefore based upon a lowest-common-denominator view of distributed actor systems. In all cases:

- actors will act sequentially, sending and receiving unidirectional, asynchronous messages;
- actors can be located on any node on the network, known only at runtime, and the system will be able to locate, though a system specific mechanism, on which machine an actor is located;
- message delivery is not guaranteed and node failures will be detected eventually.

PARTISAN follows this lowest-common-denominator view of distributed actor systems for the sake of portability of these

ideas; the same principles behind our work can be applied to realizations of PARTISAN for the other industrial-grade actor systems, such as Akka and Orleans. Applying these ideas to Akka would be straightforward, given the programming model is directly inspired by Erlang. Orleans has a slightly different programming model involving remote method invocations, but the underlying execution model is composed of unidirectional, asynchronous message sends and receives, the same as the Erlang programming model (and, extremely similar to Erlang's included RPC abstraction.)

Based on this view of actor systems, PARTISAN adds (i) the runtime selection of overlay network, and (ii) a collection of predominantly automatic latency reduction optimizations.

Latency Reduction Optimizations. PARTISAN applies the above three optimizations, *parallelism*, *named channels*, and *affinitized* scheduling (Section 4) to this lowest-common-denominator view of actor systems to achieve sometimes significant latency reduction (demonstrated in Section 6).

While some of these ideas for latency reduction have been explored in the context of networking, these optimizations are not exposed to the developer in distributed actor systems–this work is the first to do so, to the best of our knowledge.

In order to enable the application developer to directly take advantage of these optimizations when it makes sense for their application, application developers only need to specify the number of outgoing message queues (parallelism) and the types of messages that are being sent (named channels); affinitized scheduling is automatically performed by the runtime.

5.2 API

PARTISAN is designed to be a drop in replacement for Distributed Erlang, with each API command in PARTISAN providing a 1-to-1 correspondence with Distributed Erlang. The API of PARTISAN, and its corresponding calls in Distributed Erlang, is provided in Table 1 and an example of the transformation of a program from using Distributed Erlang to PARTISAN is provided in Listing 1. Performing this 1-to-1 transformation converts a Distributed Erlang application to use PARTISAN with optimizations disabled.

Like all distributed actor systems, PARTISAN's API provides both membership operations, that are used for joining/removing nodes from the cluster, and messaging operations, that are used for asynchronously sending messages. PARTISAN's programming model is both overlay-agnostic and asynchronous. Therefore, all operations return immediately and have overlay-specific behavior.

5.3 Implementation

PARTISAN is implemented as a library for Erlang and requires no modifications to the Erlang VM. This was in an effort to make PARTISAN's scalability and latency benefits immediately available to production Erlang applications with minimal changes to application code. PARTISAN is implemented in 6.7 KLOC and is available as an open source project on GitHub. This implementation of PARTISAN has several industry adopters and a growing community.

```
{partisan, [%% Enable affinity scheduling for all messages.
            {affinity, enabled},

            %% Enable parallel connections.
            {parallel, enabled},

            %% Optional: override default.
            {parallel_connections, 16},

            %% Specify available channels.
            {channels, [vnode, gossip, broadcast]},

            %% Selection of overlay.
            {membership_strategy,
             partisan_full_mesh_membership_strategy}]}.
```

Listing 3: Riak Core configuration for PARTISAN using options in Table 2 for experiments run in Section 6.2.

5.4 Configuration

Configuration options to select overlay, enable parallelism, and specify named channels are outlined in Table 2. Listing 3 demonstrates a configuration used in our Riak Core evaluation which enables parallelism, named channels, and affinitized scheduling for all messages. Users can choose to annotate message sends with a channel for targeted use of named channels and affinitized scheduling can be enabled for all messages or for an individual message; these options are demonstrated in Listing 2.

If the number of parallel connections is not specified by the user, the system will default to a reasonable value for this parameter based on the number of Erlang schedulers available. Under a default configuration of the Erlang VM, a single scheduler maps to a single vCPU. This default configuration and heuristic is discussed in detail in our experimental evaluation. (Section 6.1).

5.5 Bring Your Own Overlay

PARTISAN exposes an API for users to implement their own overlays; application developers must simply implement the `membership_strategy` interface for handling messages. PARTISAN automatically uses this membership strategy for processing incoming and outgoing messages to the system – the application developer only needs to handle internal state transitions and supplying the system with an updated list of members. PARTISAN automatically sets up required connections, serializes and deserializes messages, performs failure detection, and message forwarding. This makes it possible to implement protocols with very little code; our implementation of the full-mesh membership protocol is 152 LOC.

6 Experimental Evaluation

To evaluate PARTISAN, we designed a set of experiments to answer the following questions:

Feature	Configuration Option
Enable parallelism with default number of connections	`{parallel, enabled}`
Specify number of *N* connections to each peer	`{parallel_connections, N}`
Open *N* parallel connections for each of the named channels	`{channels, [Channel1, Channel2]}`
Enable affinitized scheduling for all messages	`{affinity, enabled}`
Specification of overlay	`{membership_strategy, MembershipStrategy}`

Table 2: PARTISAN's configuration options

- **RQ1:** What are the benefits of affinitizing actor messaging across a number of parallel TCP connections?
- **RQ2:** Can these optimizations be used on real-world applications to achieve reduction in message latencies?
- **RQ3:** Does the selection of the overlay at runtime provide better scaling properties for the application?

We begin with a set of microbenchmarks (Section 6.1), where we seek to examine the benefits of affinitizing actor communication across a number of parallel connections. We demonstrate that PARTISAN's optimizations can provide reductions in latency for workloads containing large objects, or when deployed in high latency scenarios.

Next, we examine the applicability of these optimizations on real-world applications (Section 6.2). Using a real-world distributed programming framework with an example key-value store, we show a significant reduction in latency under both high latency scenarios (datacenter-to-datacenter communication) and large object workloads through the use of a combination of optimizations: *parallelism*, *named channels*, and *affinitized* scheduling.

Finally, we explore the selection of the overlay on scaling to larger clusters (Section 6.3). We demonstrate that we can scale to order-of-magnitude larger clusters while maintaining the same application semantics by specializing the overlay at runtime to the application.

6.1 Microbenchmarks

To evaluate the optimizations in PARTISAN around latency reduction (**RQ1**), we set out to answer the following questions: (i) what is the effect of affinitizing actors; (ii) how does one know how many parallel connections to use when affinitizing actors; (iii) does affinitized parallelism benefit workloads in high latency scenarios; and (iv) does affinitized parallelism benefit workloads with large object sizes? We present a set of microbenchmarks that address each of these questions.

Experimental Setup. For the microbenchmarks, we used a single Linux virtual machine with 16 vCPUs with 64 GB of memory. On this machine, we ran two instances of the Erlang VM that communicate with one another using TCP with either a simulated RTT latency of 1ms (RTT within a single AWS availability zone) or 20ms (RTT between two availability zones in the same AWS region.) A single Linux VM is used for hosting both instances of the Erlang VM to ensure no interference from the external network and to guarantee a fixed latency during the duration of the experiment. This virtual machine is purposely kept underloaded, as to not see the effects of resource contention inside the Linux VM on latency. Each Erlang VM is configured to run 16 schedulers with kernel polling enabled.

Each of the microbenchmarks runs multiple configurations of PARTISAN under both increasing latency and payload size, with a fixed number of $10,000$ messages per actor, per experiment. We consider PARTISAN with parallelism disabled, PARTISAN with parallelism, and PARTISAN with affinitized parallelism. We do not consider named channels in the microbenchmarks, as named channels and affinitized parallelism serve the same function: partitioning communication across a number of TCP connections either automatically or by using a user-specified partitioning key.

At the start of each experiment, *N* actors are spawned on each of two instances of the Erlang VM (unless otherwise specified, as in Figure 2), based on the desired concurrency level. Each actor will send a single message to an actor on the other node and wait for acknowledgement before proceeding. Experiments were run using the *full-mesh* overlay, but the optimizations are implemented for all overlays. Latency is reported as the time to send a single message from the source to the destination.

Results. We start by showing a baseline configuration of Distributed Erlang compared with PARTISAN in Figure 1. Our results show that leveraging additional connections and affinitizing communication increases performance regardless of concurrency. With 128 actors, 512KB payload, and 1ms RTT, PARTISAN with affinitized parallelism performs 1.69x better than Distributed Erlang. Considering parallelism, but without affinity, yields a 1.90x performance improvement. With a uniform workload and without the network as a bottleneck, affinitized scheduling yields a performance benefit over Distributed Erlang, but introduces a slight performance penalty when compared to purely random scheduling.

In Figure 1, the number of parallel connections is specified as 16; however, picking this number is not necessarily trivial! Figure 2 shows the effects on outliers based on the number of connections the system needs to maintain to its peers. Here, we demonstrate that 16 connections is a good choice for connections (and, the number selected as our best case in all experiments.) But why 16? 16 is selected using the heuristic that each Erlang VM is running 16 schedulers, one mapped

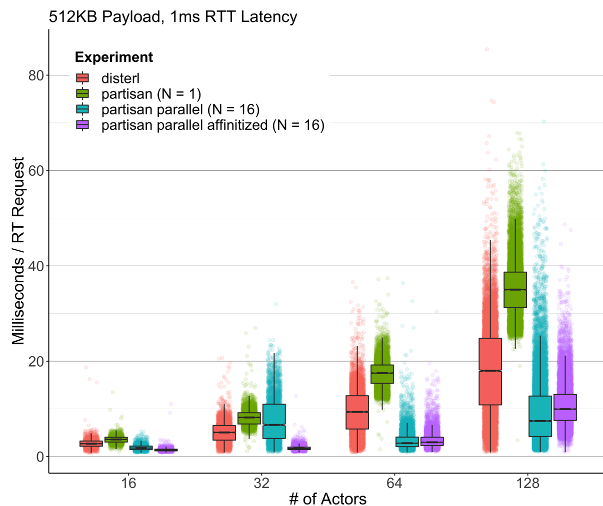


Figure 1: Performance of Distributed Erlang and PARTISAN broken out by optimization.

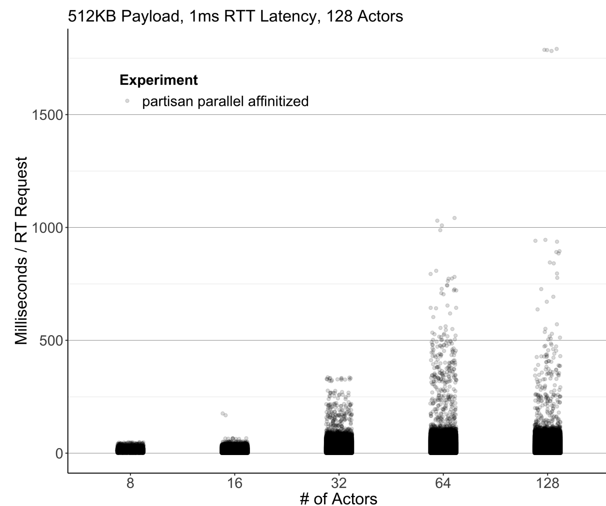


Figure 2: Effects of scaling connections with the number of actors on outliers.

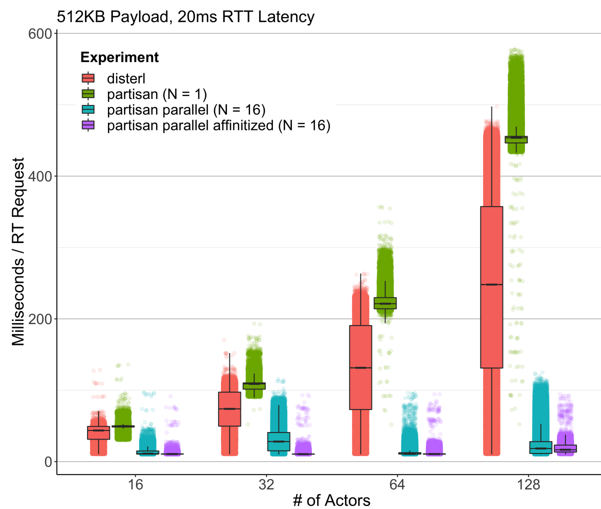


Figure 3: Performance of Distributed Erlang and PARTISAN broken out by optimization under a high latency workload: round trip time between actors is set at 20ms, object size is set at 512KB.

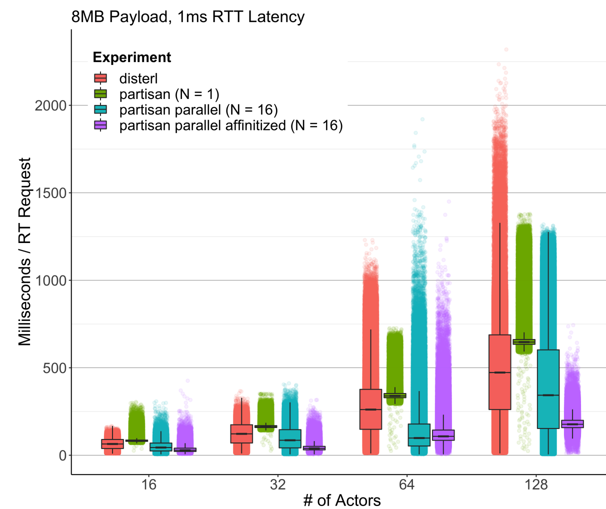


Figure 4: Performance of Distributed Erlang and PARTISAN broken out by optimization under a large payload workload: round trip time between actors is set at 1ms, object size is set at 8MB.

to a particular vCPU, and when the system needs to maintain more connections than available schedulers, context switching penalties manifest themselves as outliers (shown in Figure 2).

Focusing on these outliers, we might ask how bad does it get? With 128 actors, 512KB payload, and 1ms RTT, moving from 16 connections to 128 connections increases outliers from a max value of 176ms to 1791ms, a 10.17x increase!

In Figure 3, we turn our attention to the question of network conditions. In our first experiment (Figure 1), we chose a 1ms RTT to explore performance in a scenario where we can assume our application is running within a single AWS availability zone. But what happens if we don't have such favorable network conditions? What if our application is spread out between two AWS availability zones and suffers from RTTs closer to 20ms instead? Figure 3 shows the effects of running our earlier experiment this time with a 20ms RTT latency between actors located on different nodes. As we can see, as the latency increases, the system can take advantage of more communications channels to parallelize inter-actor communication on the network. With 128 actors, 512KB payload, and 20ms RTT, PARTISAN with parallelism performs 10.92x better than Distributed Erlang. By affinitizing parallelism, per-

formance increases to 13.50x better than Distributed Erlang.

In the Erlang community, large message sizes are not uncommon. Consider again Riak, the distributed key-value store which could contain user-stored and arbitrary-sized data. An Erlang message then could contain a user-provided piece of data megabytes in size. However, it's well-known in the Erlang community that Distributed Erlang doesn't handle large message sizes well. In fact, the Riak documentation suggests to avoid storing objects larger than 1-2MB due to the performance degradation that occurs due to Distributed Erlang [5,15]. Cognizant of this, we turn our attention to question of how large payload size affects performance in PARTISAN. Can PARTISAN overcome some of the performance issues faced by Distributed Erlang with large payloads?

Figure 4 explores the effects of increasing payload size on PARTISAN as compared to Distributed Erlang. Keeping in line with the community-observed limits of Distributed Erlang, we vary the message size from 512kb (below the 1MB performance degradation threshold) to 8MB (far above the 1MB performance degradation threshold). With 128 actors, 8MB payload, and 1ms RTT, PARTISAN with parallelism performs 1.20x better than Distributed Erlang! By affinitizing parallelism, performance increases to 2.63x.

Discussion. So far, we've seen that PARTISAN outperforms Distributed Erlang in all of our microbenchmarks. We've shown that the collection of optimizations made available to Erlang applications by PARTISAN (that is, leveraging additional connections, and affinitizing work to those connections based on the type of message and the node that the message is being sent to), can drastically improve performance by reducing latency, in some cases by over 30x.

But what does this mean practically? From these experiments, it's clear that Distributed Erlang was designed when the sort of applications being written was limited as compared to what we would like to write today; i.e., applications that send small payloads within a single data center.

As we have shown in these experiments, PARTISAN goes beyond this, and seems to be well-suited for enabling new types of applications, such as: (i) applications that operate with large data-centric workloads; (ii) applications that operate at a geo-distributed scale; (iii) the combination of both.

6.2 Evaluation: Latency Reduction in Riak

To determine the applicability of these optimizations to real-world programs (**RQ2**), we asked the following questions: (i) is it possible to modify existing application code to take advantage of the PARTISAN optimizations through the use of PARTISAN's API, and (ii) do these optimizations result in the reduction of latency for these programs?

To answer these, we ported the distributed systems framework, Riak Core, to PARTISAN and built two example applications: (i) a simple echo service – an application that's designed to only be bound by the speed of the actor receiving messages and the network itself; and (ii) a memory-based key-value store that operates using read/write quorums – more representative of a workload where more data is being transmitted and more CPU work has to occur.

6.2.1 Background: Riak Core

Riak Core is a distributed programming framework written in Erlang and based on the Amazon Dynamo [13] system that influenced the design of the distributed database Riak, Apache Cassandra, and the distributed actor framework Akka.

In Riak Core, a distributed hash table is used to partition a hash space across a cluster of nodes. These *virtual nodes*—the division of the hash space into N partitions—are claimed by a node in the cluster, and the resulting ownership is stored in a data structure known as the ring that is periodically gossiped to all nodes in the cluster. Requests for a given key are routed to a node in the cluster based on the current partitioning of virtual nodes to cluster nodes in the ring structure using consistent hashing, which minimizes the impact of reshuffling when nodes join and leave the cluster. Background processes are used for cluster maintenance; ownership handoff, (transferring virtual node ownership) metadata anti-entropy (an internal KVS for configuration metadata) and ring gossip (information about the cluster's virtual node to node mapping.)

In our experimental configuration we use 1,024 virtual nodes, the largest possible ring configuration for Riak Core. This ring size requires the largest amount of system resources – we account for this in our experiment – however, provides the most fine-grained partitioning for individual requests.

6.2.2 Modifications to Riak Core to Support PARTISAN

To perform our evaluation of PARTISAN using Riak Core, it was necessary to modify the existing application to take advantage of PARTISAN's APIs. Our changeset to Riak Core in order to use PARTISAN instead of Distributed Erlang is fairly minimal: 290 additions and 42 removals including additional logging for debugging, additional tests, and configuration.

The authors of Riak Core already realized that request traffic and background traffic could be problematic, so one mechanism inside of Riak Core—ownership handoff, responsible for moving data between virtual nodes when partitioning changes—already manages it's own set of connections. This mechanism alone contains roughly 900 LOC for connection maintenance – code that could be eliminated and replaced with calls to the PARTISAN API.

6.2.3 Echo Service

Experimental Setup. Our first application is a simple echo service, implemented on a three node Riak Core cluster. For each request, we generate a binary object, uniformly select a partition to send the request to, and wait for a reply containing the original message before issuing the next request. For each request, we draw a key from a uniform distribution over 1,024 keys – matching the ring size of the cluster – and run the key through Riak Core's consistent hashing algorithm for placement of the request. Requests originate at all of the nodes

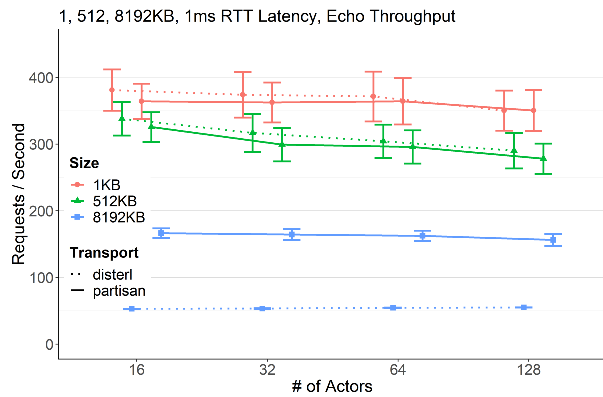


Figure 5: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the echo service / low latency workload: round trip time between actors is set at 1ms, object size varies 1, 512, and 8192KB.

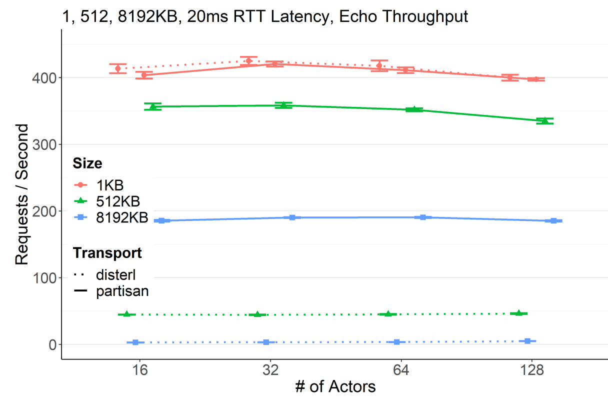


Figure 6: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the echo service / high latency workload: round trip time between actors is set at 20ms, object size varies 1, 512, and 8192KB.

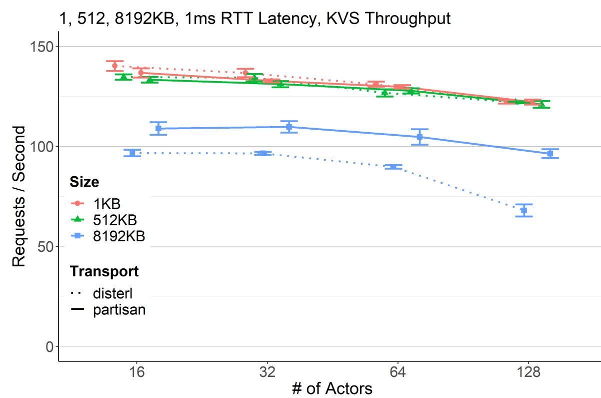


Figure 7: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the KVS / low latency workload: round trip time between actors is set at 1ms, object size varies 1, 512, and 8192KB.

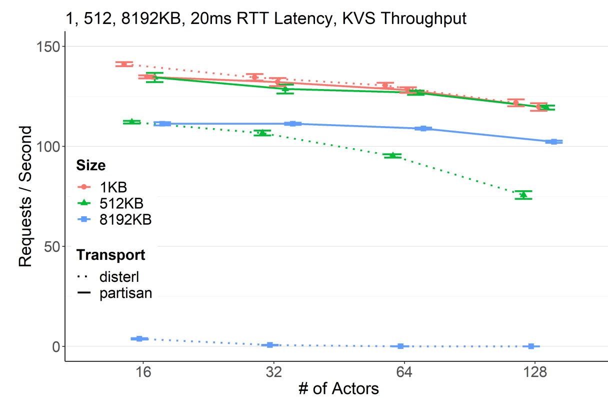


Figure 8: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the KVS / high latency workload: round trip time between actors is set at 20ms, object size varies 1, 512, and 8192KB.

in the cluster, and based on the key placement, are routed to the node responsible for handling the request. To ensure we can compare the results between runs, we wait for the cluster to stabilize before beginning the experiment.

Binary objects are generated for three payload sizes, 1KB, 512KB and 8192KB. Concurrency is increased during the test execution and parallelism is configured at 16. We test two latency configurations: 1ms, shown in Figure 5, and 20ms, shown in Figure 6. We run a fixed duration of 120 seconds.

Results. Figure 5 demonstrates that with 128 actors, 1ms RTT, and large payloads (8MB), PARTISAN is 2.84x faster than Distributed Erlang. With medium (512KB) and small payloads (1KB), PARTISAN is on par with Distributed Erlang (0.95x - 1.00x).

Figure 6 demonstrates that with 128 actors, 20ms RTT, and larger payloads (8MB), PARTISAN is 38.07x faster than Distributed Erlang (which achieves only 5 ops/second before reaching peak throughput). With medium payloads (512KB), PARTISAN is 7.25x faster than Distributed Erlang. With small payloads (1KB), PARTISAN is on par with Distributed Erlang (0.99x).

6.2.4 Key-Value Store

Experimental Setup. Our second application is a memory-based key-value store, similar to the Riak database, implemented on a three node Riak Core cluster.

Each key is hashed and mapped to a virtual node using the ring structure that is gossiped in the cluster. The virtual node that the key is hashed to, along with that virtual nodes' two clockwise neighbors on the ring, represent the three virtual nodes that contain the three replicas for the data item. Each request (either a get operation or put operation) to the key-value store uses a quorum request pattern, where requests are

made to these three replicas, and the response is returned to the user when a majority (2 out of 3) replicas reply.

This pattern involves multiple nodes in the request path, and each partition simulates a 1ms storage delay in the request path. We reuse the aforementioned benchmarking strategy: test execution is fixed at 120 seconds.

For each request, we draw a key from a normal distribution across 10,000 keys and run the key through Riak Core's consistent hashing algorithm for placement. The consistent hashing placement algorithm aims for uniform partitioning of keys across the cluster. Requests originate at all of the nodes in the cluster, and based on the key placement, are routed to the node(s) responsible for handling the request. To ensure we can compare the results between runs, we wait for the cluster to stabilize before beginning the experiment. We use a 10:1 read/write ratio for the experimental workload. Concurrency is varied in our experiments (x-axis) and parallelism is configured at 16. We test two latency configurations: 1ms, shown in Figure 7, and 20ms, shown in Figure 8.

Results. Figure 7 demonstrates that with 128 actors, 1ms RTT, and both medium (512KB) and small (1KB) payloads, PARTISAN performs on par with Distributed Erlang (0.99x-1.00x). With larger payloads (8MB), PARTISAN is 1.42x faster than Distributed Erlang.

Figure 8 demonstrates that with 128 actors, 20ms RTT, and small (1KB) payloads, PARTISAN performs on par with Distributed Erlang (0.98x). With medium payloads (512KB), PARTISAN is 1.50x faster than Distributed Erlang. With large payloads (8MB), PARTISAN far exceeds the performance of Distributed Erlang, achieving 102 ops/second; Distributed Erlang only completes 1 operation during the entire 120s execution.

6.2.5 Discussion

As we have shown in these experiments, PARTISAN is not only well-suited as a replacement for Distributed Erlang, given its similar performance under workloads that Distributed Erlang was designed for, but PARTISAN also enables new classes of applications in distributed actor frameworks. Our experiments have shown increased throughput in applications with large data-centric workloads: an example of this would be the Riak distributed database without 1MB storage limitations.

6.3 Evaluation: Improving Scalability in Lasp

In our previous experiment on latency reduction in Riak Core, we demonstrated optimizations for latency reduction in a distributed database that communicates with all of the nodes in the cluster. This is one example of an application that benefits from the *full-mesh* overlay. However, not all applications benefit from, nor require, the full-mesh model that is default case in Distributed Erlang. In this section, we address the question of whether or not an application can benefit from selection of the overlay at runtime (**RQ3**): specifically, the *client-server* and *peer-to-peer* overlays.

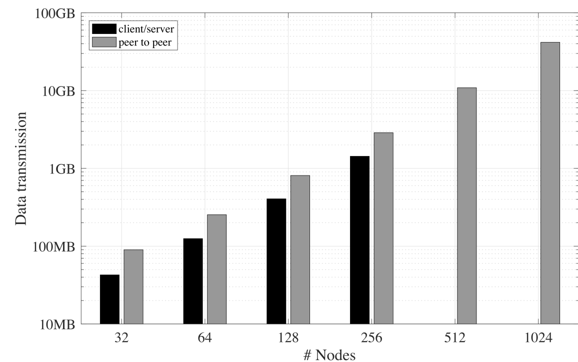


Figure 9: Comparison of data transmission for Lasp deployed on the client-server and peer-to-peer overlays for different cluster sizes (32 to 1024 nodes).

In order to understand the effect of overlay on scalability, we focus on how many nodes we can scale our application to under each overlay for an advertisement counter application implemented with Lasp.

6.3.1 Lasp

Lasp [22] is a programming framework designed for large scale coordination-free programming. Applications in Lasp are written using shared state; this shared state is stored in an underlying key-value store and is replicated between all nodes. Applications modify their own replica and propagate the effects of their changes to their peers. Lasp ensures that applications converge to the same result on every node through the use of data structures known as Conflict-Free Replicated Data Types [29], combined with monotone programming [3].

For our Lasp evaluation, the application is a simulated advertisement counter, modeled after the Rovio counter scenario for Angry Birds [22]. In this application, each client keeps a replica of a distributed counter that is incremented every time an advertisement is displayed to the user and whose state is periodically propagated to other peers in the system. When a certain number of impressions is reached, the advertisement is disabled and no longer displayed to the user.

The distributed counter used was a particular type of CRDT: a Grow-Only Counter (G-Counter). The G-Counter maps node identifiers at each of the clients to a monotonically increasing counter. Clients increment their position in the map and when merging state propagated from other nodes in the system, the pair-wise maximum is taken for each component in the map. To determine when an advertisement can be disabled, a lower bound is checked according to the sum of the components in the map: this represents a lower bound on the total number of times an advertisement has been displayed.

Experimental Setup. For this evaluation, a total of 70 m3.2xlarge Amazon EC2 instances in the same region and availability zone. Mesos [16], is used to subdivide each of these machines into smaller, fully-isolated machines. Each

container in Mesos represents a single Lasp node that communicates with other nodes in the cluster using PARTISAN.

The increment interval for each counter was fixed at 10s, and the propagation interval for the counter was fixed at 5s. The total number of impressions was configured to ensure that the experiment would run for 30 minutes under all configurations. The evaluation is performed on both the *client-server* and *peer-to-peer* overlays for different cluster sizes, ranging from 32 all the way up to 1,024 node clusters. For both overlays, the system propagates the full state of the counter to the node's peers at each propagation interval.

Note that since the Rovio advertisement counter scenario was designed for mobile applications, we do not run the full-mesh topology because it would be unrealistic. That is, in the context of mobile apps, clients would not connect to all other nodes, nor will they have knowledge of who all of the clients in the system are. Rather, either mobile apps will communicate with some number of nearby peers (peer-to-peer) or they will communicate through a server (client-server). Client-server also serves as the standard model of deploying mobile applications today. Thus, we designed our experiments to reflect this—we examine client-server and peer-to-peer overlays for this application in our experiments.

Results. Figure 9 presents the total data transmission required for the experiment to finish as we scale the size of the cluster from 32 to 1024 nodes. For smaller clusters of nodes, client-server is the more efficient overlay in terms of the amount of data that must be transmitted to finish the experiment. However, this improved efficiency comes at a cost: the client-server configuration is unable to scale beyond 256 nodes. More specifically, the experiment fails to complete because of a crash failure of the server. This crash failure occurs because of unbounded message queues: when the server is unable to process the incoming messages from the clients quickly enough, the Erlang VM allocates all available memory for storage of the message queue. This unbounded allocation results in termination of the Erlang by the Linux OOM killer once the instance runs out of available memory.

Peer-to-peer is more resilient in the face of a node failure allowing it to support larger clusters of nodes—up to 1024! However, peer-to-peer is less efficient due to this—the redundancy of communication links used by the overlay causes it to transmit more data in order to complete the experiments.

Discussion. Perhaps the most interesting takeaway from the results of this real-world large-scale experiment is that the experiment was even possible at all with Erlang. As Distributed Erlang permits one to only use a full-mesh overlay, it's possible that the previous results observed by Ericsson [1] on the maximum size of Erlang clusters–only 200 nodes–are due to this full-mesh-only restriction.

This experiment suggests that PARTISAN may enable the development of new applications with actors systems that have not been previously possible by enabling the application developer to, at runtime, change the pattern of communication between nodes, without altering application semantics. Perhaps the lack of mobile applications or even IoT applications written using distributed actor systems is a symptom of the full-mesh-only restriction.

7 Related Work

Head-of-line blocking is a well-known issue in the systems and networking community, especially in systems that use multiplexed connections. Facebook's TAO [9] relies on multiplexed connections but allows out-of-order responses to prevent head-of-line blocking issues. Riak CS [7], an S3-API compatible object storage system build on Riak, arbitrarily chunks data into 1MB segments to prevent head-of-line blocking. Geo-replicated Riak [6] contains an ad hoc implementation of node-to-node messaging to avoid Distributed Erlang at cross-region latencies. Distributed Erlang now includes a feature for arbitrarily segmenting messages into smaller chunks to reduce the impact of head-of-line blocking [17].

Ghaffari *et al.* [15] identified several factors limiting Erlang's scalability: (i) increasing payload size and (ii) head-of-line blocking with Erlang's RPC mechanism – two of the limiting factors in Riak 1.1.1's ≈ 60 node limit on scalability. Chechina *et al.* [11] proposed partitioning the graph of nodes into subgraphs and using supernodes for connecting the groups, avoiding the problems of full-mesh connectivity.

8 Conclusion

We presented PARTISAN, an alternative runtime system for improved scalability and reduced latency in actor applications. PARTISAN provides higher scalability by allowing the application developer to specify the network overlay used at runtime without changing application semantics, thereby specializing the network communication patterns to the application. PARTISAN reduces message latency through a combination of three predominately automatic optimizations: *parallelism*, *named channels*, and *affinitized* scheduling. We implemented PARTISAN in Erlang and showed that PARTISAN achieves up to an order of magnitude increase in the number of nodes the system can scale to through runtime overlay selection, up to a 38.07x increase in throughput, and up to a 13.5x reduction in latency over Distributed Erlang.

Acknowledgments

We would like to thank Scott Fritchie, Zeeshan Lakhani, Frank McSherry, Jon Meredith, Andrew Stone, Andrew Thompson, the anonymous reviewers, and our shepherd Ryan Stutsman, for their valuable feedback on this paper.

Availability

PARTISAN is available at https://github.com/lasp-lang/partisan. Instructions for reproducing our results are available at https://github.com/cmeiklejohn/partisan-usenix-atc-2019.

References

[1] Ericsson AB. Personal communication.

[2] Octavo Labs AG. Vernemq. https://vernemq.com. Accessed: 2018-02-03.

[3] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[4] Apache. Couchdb. http://couchdb.apache.org. Accessed: 2018-02-03.

[5] Basho Technologies, Inc. Developing with Riak KV. https://docs.basho.com/riak/kv/2.1.1/developing/faq/. Accessed: 2019-01-19.

[6] Basho Technologies, Inc. Riak. https://github.com/basho/riak. Accessed: 2018-02-03.

[7] Basho Technologies, Inc. Riak cs. https://github.com/basho/riak_cs. Accessed: 2018-02-03.

[8] Philip A Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, et al. Geo-distribution of actor-based services. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):107, 2017.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[10] 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.

[11] Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin, and Robert Virding. The design of scalable distributed erlang. In *Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK*, page 85, 2012.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[13] 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 *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[14] Malcolm Dowse. Erlang and First-Person Shooters. http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf. Accessed: 2018-09-26.

[15] Amir Ghaffari. Investigating the scalability limits of distributed erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 43–49. ACM, 2014.

[16] 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.

[17] Kenneth Lundin. Erlang latest news. http://erlang.org/workshop/2018/. Erlang Workshop 2018.

[18] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[19] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 301–310. IEEE, 2007.

[20] Joao Leitao, Jose Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 419–429. IEEE, 2007.

[21] Lightbend. Akka cluster documentation. https://doc.akka.io/docs/akka/2.5/index-cluster.html. Accessed: 2018-02-03.

[22] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195. ACM, 2015.

[23] Netflix. Atlas. https://github.com/Netflix/atlas. Accessed: 2018-10-01.

[24] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 38. ACM, 2016.

[25] Pivotal. Rabbitmq. https://www.rabbitmq.com. Accessed: 2018-02-03.

[26] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, 2009.

[27] Riot Games. Chat Service Architecture: Persistence. https://engineering.riotgames.com/news/chat-service-architecture-persistence. Accessed: 2018-09-26.

[28] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Communications of the ACM*, 53(10):72–82, 2010.

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[30] Randall Stewart and Chris Metz. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, (6):64–69, 2001.

[31] Claes Wikström. Distributed programming in erlang. In *In PASCO'94-First International Symposium on Parallel Symbolic Computation*. Citeseer, 1994.

Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation

Changheng Song[†], Wenwen Wang[‡], Pen-Chung Yew[‡], Antonia Zhai[‡], and Weihua Zhang[†]
[†]*Software School, Fudan University*
[†]*Shanghai Key Laboratory of Data Science, Fudan University*
[‡]*Department of Computer Science and Engineering, University of Minnesota, Twin Cities*
[†]{17212010032, zhangweihua}@fudan.edu.cn, [‡]{wang6495, yew, zhai}@umn.edu

Abstract

Dynamic binary translation (DBT) is a key system technology that enables many important system applications such as system virtualization and emulation. To achieve good performance, it is important for a DBT system to be equipped with high-quality translation rules. However, most translation rules in existing DBT systems are created manually with high engineering efforts and poor quality. To solve this problem, a learning-based approach was recently proposed to automatically learn semantically-equivalent translation rules, and symbolic verification is used to prove the semantic equivalence of such rules. But, they still suffer from some shortcomings.

In this paper, we first give an in-depth analysis on the constraints of prior learning-based methods and observe that the equivalence requirements are often unduly restrictive. It excludes many potentially high-quality rule candidates from being included and applied. Based on this observation, we propose an enhanced learning-based approach that relaxes such equivalence requirements but supplements them with constraining conditions to make them semantically equivalent when such rules are applied. Experimental results on SPEC CINT2006 show that the proposed approach can improve the dynamic coverage of the translation from 55.7% to 69.1% and the static coverage from 52.2% to 61.8%, compared to the original approach. Moreover, up to 1.65X performance speedup with an average of 1.19X are observed.

1 Introduction

Dynamic binary translation (DBT) is a key enabling technology for many critical system applications such as system virtualization and emulation [20, 28], whole program/system analysis [6, 13], software development and debugging [14], security vulnerability detection and defense [15, 17], computer architecture simulation [22, 27, 29], and mobile computation offloading [26]. There have been many widely-used DBT systems, such as Pin [18], Valgrind [21] and QEMU [2].

In general, a DBT system takes an executable binary code in one instruction set architecture (called *guest* ISA) and dynamically translates it into the binary code in another instruction set architecture (called *host* ISA). The translation process is mostly driven by *translation rules* that translate guest instructions into a sequence of semantically-equivalent host instructions [23].

For a DBT system, its performance is dominated by the quality of the translated host binary code [25]. Therefore, it is very important for a DBT system to be equipped with high-quality translation rules. However, due to the complexity and opacity of modern ISAs, it is difficult to *manually* construct such high-quality translation rules as it poses a significant engineering challenge. Even worse, to support retargetable DBTs (from multiple guest ISAs into multiple host ISAs) in the same framework, a set of pseudo-instructions are commonly used as their internal representations [2]. As the execution time is directly proportionate to the number of host instructions executed, such a multiplying effect has a significant impact on the overall DBT performance.

To improve the quality of translation rules and reduce engineering efforts, a learning-based approach [23] is recently proposed to learn automatically binary translation rules. Since the translation rules are learned from the optimized binary codes generated by the compiler, this approach is capable of yielding higher quality translation rules than existing manual schemes. Moreover, the whole learning process can be fully automated without manual intervention. Although the above approach is attractive, it still suffers from some fundamental limitations. That is, a translation rule can be harvested (i.e., learned) only if the guest and the host binary code that correspond to the same program source statement(s) are strictly semantically equivalent. This is enforced through a symbolic verification process.

On the surface, this equivalence verification process is necessary and appropriate because it guarantees the *correctness* of the learned rules. However, further investigation reveals that this equivalence requirement is often unduly restrictive. It excludes many potentially high-quality rule candidates from being harvested and applied. In particular, such restrictions usually keep architecture-specific instructions in guest and/or

host ISAs from being included for more efficient translation as they are mostly architecturally specific, and thus inherently different and more challenging to prove semantic equivalence.

To overcome this limitation, this paper presents an enhanced learning-based approach that relaxes such restrictions and allows more translation rules to be harvested and applied. More specifically, it purposely relaxes the requirements of semantic equivalence and allows semantic discrepancies between the guest and host instructions to exist in the translation rules, e.g., different condition codes or the different number of operands in matching guest and host instructions. Symbolic verification process is no longer just to check the *strict* semantic equivalence between the matching guest and host instructions, but also to identify the specific semantic discrepancies between them that can be used during the rule application phase to verify whether such discrepancies either will not cause ill effect, or are satisfied in the context of the rules being applied (for more details see Section 4). We call such semantic equivalence in the translation rules *constrained semantic equivalence* as the specific semantic discrepancies of the translation rules become the *constraining conditions* for such rules to be safely applied. This requires some runtime program analysis (mostly in a very limited scope) during the rule application phase, which usually incurs very small overhead. Those with very complicated constraining conditions that require extensive runtime program analysis will be discarded.

To demonstrate the feasibility and the benefit of such constrained-equivalent translation rules, we have implemented a prototype based on the proposed approach. The prototype includes an enhanced learning framework and a DBT system that applies the constrained-equivalent translation rules to generate host binary code. We evaluate the implemented prototype using SPEC CINT2006. Experimental result shows that the proposed approach can significantly improve the harvest rate of the learning process from 20.3% to 25.1% and dynamic coverage from 55.7% to 69.1% while static coverage from 52.2% to 61.8%, compared to the original learning approach in [23]. Moreover, no degradation is observed for the learning efficiency, i.e., around 2 seconds to yield a translation rule, which is the same as the original learning process. After applying the enhanced translation rules, we achieve up to 1.65X performance speedup with an average of 1.19X compared to the original approach.

In summary, this paper makes the following contributions:

- We propose an enhanced learning-based approach that can harvest and apply constrained-equivalent translation rules discarded by the original approach, and allows DBT systems to generate more efficient host binary code.
- We implement the proposed learning-based approach in a prototype, which includes a learning framework based on LLVM and a DBT system extended from QEMU to accept the constrained-equivalent translation rules.
- We conduct some experiments to evaluate the proposed learning-based approach. Experimental results on SPEC CINT2006 shows that our approach can achieve up to 1.65X speedup with an average of 1.19X compared to the original learning approach.

The rest of this paper is organized as follows. Section 2 presents some background of the original non-constrained semantically-equivalent learning-based approach. In Section 3, we identify some technical challenges in learning and applying constrained-equivalent translation rules. Section 4 presents the design issues of our enhanced learning-based approach. In Section 5, we describe some implementation details of the prototype and evaluate the proposed approach and show some experimental results. Section 6 presents some related work and Section 7 concludes the paper.

2 Background

In this section, we introduce some background information on how a DBT and a learning-based approach such as the one proposed in [23] work.

2.1 Dynamic Binary Translation (DBT)

Typically, a DBT system adopts a guest *basic block* (or *block* for short) as the translation unit to translate guest binary code into host binary code. A basic block comprises a sequence of instructions with only one entry and one exit, and thus whenever the first instruction of a basic block is executed, the rest of the instructions in this block will be executed exactly once in order. It is worth noting that, due to the semantic differences between the guest and host ISAs, one guest block may be translated into multiple host blocks by the DBT system.

To translate a guest basic block, the DBT system firstly disassembles the guest binaries to obtain guest assembly instructions. Then, it tries to match the guest instructions with available *translation rules*. After a matched translation rule is found, the corresponding guest instructions are translated into host instructions as specified in the translation rule. This process could be iterated multiple times until all instructions in the guest block are translated. Finally, the generated host instructions are assembled into host binaries and executed directly on host machines. Figure 1 shows an example of such a translation process, where ARM is the guest ISA, and x86 is the host ISA. In this example, two translation rules are applied to translate two ARM instructions into two x86 instructions, respectively.

To mitigate the performance overhead incurred during the translation process, especially for short-running guest applications, the translated host binary code is stored into a memory region called *code cache*, and reused in the later execution. After all instructions in a guest block are translated, the execution flow of the DBT system is transferred to the code cache.

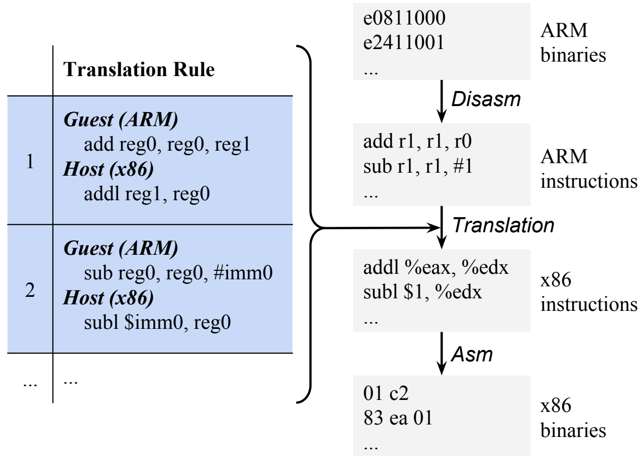


Figure 1: Dynamic binary translation from ARM to x86 driven by manually-constructed translation rules. Here, for simplicity, we assume the guest registers `r0` and `r1` are emulated using the host registers `eax` and `edx`, respectively.

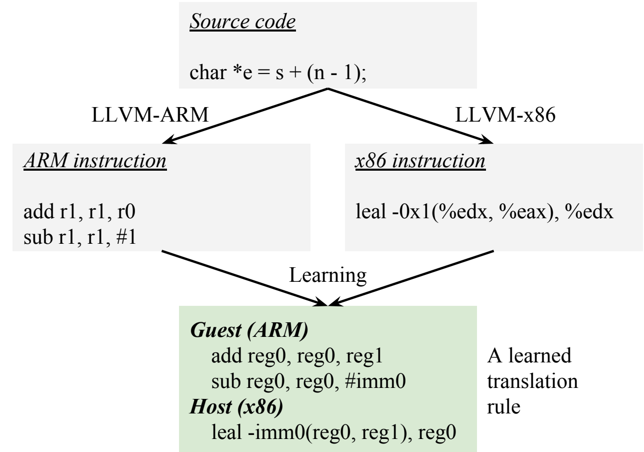


Figure 2: Automatically learning binary translation rules during the compilation process of program source code. Compared to the translation rules used in Figure 1, the learned translation rule can generate more efficient host binary code.

A *hash table* is employed to establish the mapping between the guest binary code and the corresponding translated host binary code in the code cache. Each time a guest block is encountered, the hash table is looked up to find out whether there exists a host code in the code cache that corresponds to this guest block. If yes, the translation process will be skipped, and the stored host binary code will be executed. Otherwise, the guest block is translated, and the hash table is updated with the added translated host binary.

2.2 Learning Translation Rules

As mentioned earlier, the translation process in a DBT system is mainly directed by translation rules, which also determine the quality (i.e., performance) of the translated host binary code. Therefore, it is vital for a DBT system to have high-quality translation rules for better performance. However, in practice, it is a significant engineering challenge to develop high-quality translation rules as most translation rules in existing DBT systems are constructed manually by developers. Moreover, modern ISAs are often documented in obscure and tediously long manuals. For example, Intel's manual has around 1500 pages for the x86 ISA. It requires substantial engineering efforts to understand both the guest and the host ISAs to construct high-quality translation rules.

To solve this problem, a recent approach proposes to automatically learn binary translation rules [23]. More specifically, this approach uses the same compiler for different ISAs, i.e., LLVM-ARM and LLVM-x86, to compile the same source program. During the compilation process, it extracts binary translation rules from ARM and x86 binary code that correspond to the same program source statement(s). This is inspired by the observation that the binary code compiled by the same compiler for different ISAs from the same program source code should be equivalent in program semantics. To further enforce such equivalence requirement, a symbolic verification engine is developed to filter out rule candidates in which guest and host binary code are not semantically equivalent.

Figure 2 illustrates an example of the above learning process, In this example, the program source statement is compiled into two ARM instructions and one x86 instruction by LLVM-ARM and LLVM-x86, respectively. Using symbolic execution, we can verify that the guest ARM register `r1` and the host x86 register `edx` should have the same value assuming the same initial condition. We can thus prove that the sequence of the two ARM instructions is semantically equivalent to the single x86 instruction in the example. A translation rule that maps the sequence of the two ARM instructions into one x86 instruction can then be harvested. Recall the example in Figure 1. If we use this learned rule to translate the guest binaries, we only need one host instruction instead of two as shown in the example, i.e. more efficient host binary code can be generated.

3 Issues and Challenges

The significance of the above learning approach is two folds. Firstly, it can automatically learn binary translation rules for DBT systems with less burden on developers. Secondly, given that the translation rules are learned directly from binary code generated by the native compilers, it is more likely that the harvested translation rules are more optimized than the translation rules naïvely constructed by hand, as shown in Figure 2 and Figure 1.

Theoretically, if we keep training such a learning-based

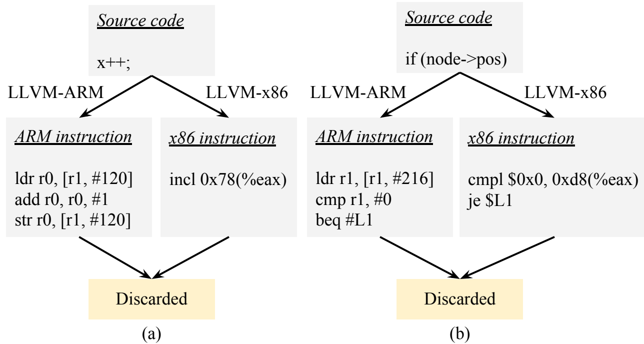


Figure 3: Two examples to demonstrate the limitation of the learning approach in [23]. These two rule candidates are discarded because the guest registers `r0` in (a) and `r1` in (b) have no equivalent host register.

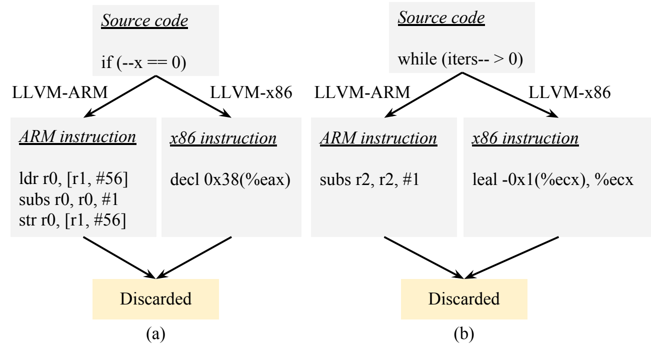


Figure 4: Another two rule candidates discarded by the learning approach in [23] because of the different condition codes in guest and host ISAs.

system with a large number of source programs, we should be able to harvest a large number of translation rules and apply them to guest binaries with good coverage. Unfortunately, after a more thorough study of this approach, we found it suffers from a fundamental limitation that prohibits it from harvesting many high-quality translation rules. In this section, we explain in more details such limitations and identify some technical challenges if we want to overcome them.

A Fundamental Limitation. To guarantee the correctness of the learned translation rules, it employs a symbolic verification engine to check the exact semantic equivalence between the guest and host binary code sequences. More specifically, the semantic equivalence is verified in three aspects that include matching register operands, memory operands and branch conditions. More details can be found in [23].

If the verification results show that the guest and host binary code sequences are not strictly equivalent, the rule candidate is discarded and no translation rule is harvested. Undoubtedly, such a verification process is necessary and appropriate. However, by a more detailed study on the discarded rule candidates, we found that the requirement of *exact* semantic equivalence is too restrictive. Many high-quality rule candidates are forced to be discarded, especially those guest and host binary code sequences that are more architecturally specific and their ISAs are significantly different, such as ARM (a reduced instruction set computer (RISC)) and Intel x86 (a complex instruction set computer (CISC)) in our example.

Figure 3 shows two examples of this limitation. Here, similar to the previous examples, the guest ISA is ARM and the host ISA is Intel x86. In Figure 3(a), the value of the variable `x` is increased by one through the increment operator as shown in the source code. With its RISC ISA, the ARM compiler generates three instructions for this source statement: loading the original value of `x`, performing the addition, and then storing the result back to `x`. In contrast, the Intel x86 compiler needs only one instruction, `incl`, with its CISC ISA.

Similarly, in Figure 3(b), the x86 instruction `cmpl` can have a memory operand, but an ARM `ldr` instruction is required before the `cmp` instruction. In these two cases, the verification will fail because there is a mismatch of register operands between the guest and the host code sequences, i.e. there is no host register that matches and holds the same value as the guest register `r0`.

However, if we examine these two examples more carefully, we will find that the root cause of the failed verification stems from the inherent differences between the guest and the host ISAs. In practice, such architectural differences are quite common and pervasive in different ISAs, even if they are both RISCs or CISCs. For instance, a post-indexed memory load instruction in ARM will modify the address register after the loading operation, while there is no similar instruction in MIPS, which is another representative RISC ISA.

In fact, these differences represent the essence of the architectural design unique to each ISAs. It is indeed a huge loss for a learning-based approach to discard such rule candidates simply because of their ISA differences. As they are architecturally specific, they are often the most efficient code sequences selected by the native compilers for specific program contexts and thus have a high potential to turn into high-quality translation rules.

Another shortcoming resulted from the aforementioned limitation is that it also excludes many rule candidates that contain instructions associated with architecture-specific hardware support. For instance, many architectures have condition codes (also known as *eflags* in x86 machines). They are single-bit registers used to store the execution summary of an instruction and can influence the control flow of the later instructions. In particular, ARM has four condition codes: negative (NF), zero (ZF), carry (CF), and overflow (VF), while x86 has seven condition codes: carry (CF), parity (PF), adjust (AF), zero (ZF), sign (SF), direction (DF), and overflow (OF).

Figure 4 shows two examples with instructions related to condition codes. In Figure 4(a), the source code decreases the

value of `x` by one and then checks the result to see whether it is zero or not. An ARM instruction `subs` is generated to perform the subtraction and update the condition codes. Here, `subs` updates all four ARM condition codes, including CF. Similarly, an x86 instruction `decl` is used to decrease the value stored in the memory operand by one and update the condition codes. However, `decl` updates all x86 condition codes, except CF. As a result, the verification process in the original learning-based approach will consider the ARM and x86 code are not semantically equivalent and discard this rule candidate. Similarly, the rule candidate in Figure 4(b) is also discarded because the x86 instruction `leal` does not update any condition code. In fact, the source code in Figure 4(a) only needs to check whether the result is zero or not, which only requires the condition code ZF. Thus, it is unnecessary to update the condition code CF, as it is never used in this context. That means, it is still possible to harvest this translation rule and apply it, if the ARM condition code CF is not used (i.e. *dead*) in the later code before it is updated. Similarly, the rule candidate in Figure 4(b) can also be harvested.

Technical Challenges. Although such limitations could exclude many high-quality translation rules during the learning process, it faces several technical challenges if we want to harvest them and apply them in a DBT system for a better performance and higher coverage.

First, we have to relax the original verification objectives as they are designed to verify the *exact* equivalence between the guest and host code sequences.

Second, given that most of those translation rules are not strictly equivalent, it is imperative that we have a mechanism to enforce their correctness when we apply them. Equally important is that the performance overhead incurred by such enforcement should be less than the performance gain they can provide.

Last but not least, in the original learning approach, a learned translation rule only needs to include two parts, i.e., the guest and host instructions, and this is typically sufficient for a DBT system. However, for the constrained-equivalent translation rules, whether we can apply these rules at runtime or not depends on the context they are being applied. As a result, we need to extend the structure of translation rules to include such constraining requirements.

4 An Enhanced Learning-Based Approach

In this section, we present the design of the proposed enhanced learning-based scheme, starting with an overview of the system framework.

4.1 Overview

The major goal of our enhanced learning-based approach is to learn and apply high-quality translation rules excluded by the original learning approach. These translation rules contain constrained-equivalent guest and host instructions, and thus cannot be harvested using the original learning approach. To this end, we redesign the learning process, reorganize the structure of the learned translation rules, and make necessary extensions to the DBT system to allow the application of the constrained-equivalent translation rules.

Figure 5 illustrates the workflow of our enhanced learning-based approach. To learn translation rules, we also compile the same program source code using the same compiler for guest and host ISAs to generate two versions of the binary code. We then extract guest and host code sequences that correspond to the same *learning scope* and consider them as the candidates for the translation rules. The learning scope is defined at the program source code level. In the original learning system, the default learning scope is set to be one source statement. The extracted guest and host code sequences then form a *rule candidate*. For each rule candidate, the next step is to verify whether the corresponding guest and host code sequences are *constrained equivalents* or not. If yes, a translation rule can be harvested. Otherwise, the rule candidate is discarded.

As an example to demonstrate our approach and by studying the rule candidates discarded by the original learning scheme, we consider the guest and host code sequences in a rule candidate as *constrained equivalent* if every modified guest *storage operand* contains the same value as a modified host storage location at the end of the code sequences and vice versa. Here the *storage operand* is broadly defined, as it can be either a register, a memory location, or a condition code (i.e., eflag). Furthermore, it is allowed that there is no corresponding modified storage location in the *host* code sequences, e.g., a corresponding condition code as mentioned earlier.

Using this relaxed and constrained equivalence definition, the guest and host code sequences can be semantically equivalent only if all *modified* guest storage operands without the corresponding host storage operands (e.g., condition codes) are not used in the following guest binaries before they are modified again. These modified guest storage operands without the corresponding host storage operands can be considered as the *constraining condition* of this constrained-equivalent translation rule.

In our framework, the semantic equivalence can be relaxed in other ways as long as the *discrepancies* can be identified and shown either having no ill effect in the context they are applied or can be compensated to make them semantically equivalent when they are applied. In other words, their *constraining conditions* can be identified and satisfied when these rules are applied. To simplify our prototype design, we only consider relaxing the requirement of exact mapping of the storage operands as defined earlier. The identified constraining conditions are integrated into the learned translation rules to determine whether it is safe to apply them or not. It is worth noting that for strictly equivalent translation rules the constraining condition is null.

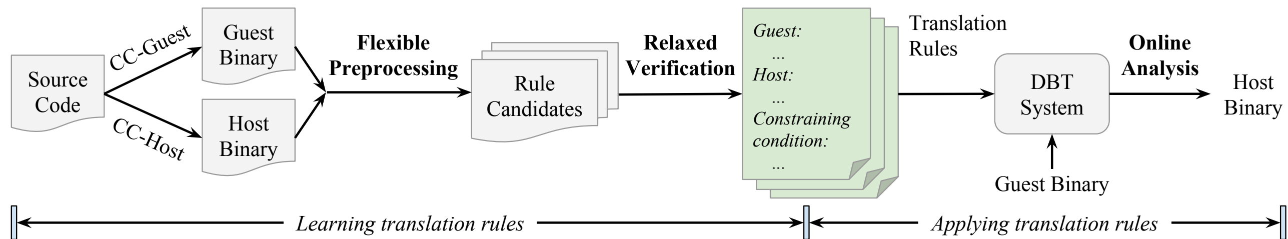


Figure 5: The work flow of the proposed enhanced learning-based approach.

To determine whether the constraining condition is met or not, a lightweight runtime analysis is employed to determine the program context in which the guest instructions is to be translated. In our case, the program context includes the information about which guest storage operand is modified by the guest instructions and used in the later code before it is modified again. The program context is then used to verify whether the constraining condition is satisfied or not. If yes, the translation rule can be applied, otherwise, it is discarded.

4.2 Varying Learning Scopes

In the original learning scheme, the learning scope is limited to one source statement. Although it appears to be reasonable, it may miss potential rule candidates as it is very common for compilers to perform optimization across multiple source statements. Therefore, our enhanced learning approach varies the learning scope from one to n source statements, and apply each learning scope over the source program.

More specifically, a *sliding window* is employed. The sliding window of size n covers n contiguous source statements, i.e. the sliding window covers the learning scope of n statements. Guest and host instructions that correspond to the n statements in this window are extracted as a rule candidate. The sliding window moves from the first line of the source code toward the end of the code. The window size is initially set to 1, and incremented by one after each pass through the sources code. When the window of size i is moved through all the source code, the number of rules learned from current window size will be compared to the number of rules learned from window sizes 1 to $i-1$. If new rules learned from window size i are less than 10% of all learned rules from window sizes 1 to $i-1$, the learning process will be stopped.

4.3 Learning Constrained-Equivalent Rules

To verify the constrained equivalence of the guest and host instructions in a rule candidate, we use the same symbolic verification engine, but relax the requirements for semantic equivalence.

First, we establish an *initial mapping* between guest and host live-in operands the same way as the original learning approach, i.e., guest registers $\mapsto$ host registers, guest memories $\mapsto$ host memories, and guest immediate values $\mapsto$ host immediate values (i.e. constants). Then, we initialize the mapped guest and host operands with the same symbol values and symbolically execute the guest and host code sequences, respectively. After the symbolic execution, we extract the symbol results of the modified guest and host registers, memories, and condition codes. These results are then fed into a SMT solver to figure out, for each modified guest register/memory/condition code, whether there exists a modified host register/memory/condition code corresponding to it or not. If each modified guest operand is mapped to a modified host operand, an original rule is generated.

If the SMT solver indicates that there exists a modified guest memory operand that does not have a matching host memory operand, we discard this rule candidate. If a modified guest register/condition code has no matching modified host register/condition code, we can harvest this rule candidate as the guest and host instructions can still be constrained equivalent. Moreover, such unmatched guest registers/condition codes are recorded as constraining conditions of the learned rules and will be checked when the rules are applied. The reason for discarding candidate rules with unmatched memory operands is that the resulting constraining conditions will require time-consuming data dependence analysis to determine whether the constrained equivalence is satisfied or not when such rules are applied.

Otherwise, all other rules are considered as a non-equivalent rule and be discarded.

4.4 Lightweight Online Analysis

For each constrained-equivalent rule to be applied, an online analysis is invoked to analyze the program context of the guest code sequence. The context information includes the data flow of the guest registers and condition codes, which can be obtained by statically analyzing the guest instructions. The context information is then used to determine whether the constraining condition of the matched constrained-equivalent rule is satisfied. For instance, if the analysis shows that a modified guest register is not mapped to any modified host register in the rule, and this modified guest register is not used in the following guest code, we can determine that the constraining condition has been satisfied in the program context, and the

translation rule can be applied.

In general, to collect the context information, the online analysis examines guest instructions that are executed after the matched guest instruction sequence. Each instruction is examined to see whether it defines or uses the register(s) or condition code(s) specified in the constraining condition of the matched translation rule. If a definition can be found before usage on *all* paths following the matched guest code sequence, the matched rule can be applied safely. Otherwise, if usage is found, the matched rule should not be applied as the modified guest register/condition code is used but the matched rule does not update it.

For indirect branch instructions, it is quite difficult to identify all possible branch targets statically. For simplicity, we stop the online analysis when an indirect branch is encountered and the translation rule will not be applied for safety consideration.

4.5 Handling Predicated Instructions

Predicated instructions are very common in many ISAs, e.g., ARM and MIPS. A predicated instruction executes only if its predicate bit is "True". Otherwise, the instruction is a "nop". For example, "add ne r0, r0, r1" in ARM will be executed only when the condition code is not equal ("ne"). Some ISAs like x86 do not support predication, and conditional branches are used instead. It is worth noting that the original learning approach cannot handle predicated instructions. So how to efficiently support predicated instructions is another important design issue for a learning-based approach because the predicate tag in predicated instructions and conditional branch are not equivalent although the execution results are the same.

In translation, we use a lightweight analysis to divide predicated instructions into multiple blocks and generate conditional branches around those blocks according to their predicate information to support the translation of predicate instructions. Before translating a basic block, we first check the predicated condition of all instructions and divide the basic block into multiple *condition blocks*. Each condition block includes instructions with the same predicated condition. In one condition block, the translation rules can be directly applied without considering the predicated condition. After a condition block is translated, a branch instruction with the opposite condition is added to the host basic block before the translated condition block is added to the host block. The branch target is the instruction following the end of the host block. This analysis is very lightweight and each basic block only needs to be checked once.

Note that an instruction with a predicated condition may change the condition codes itself. For example, `cmp ne r0, 0` will update the condition code if the last condition code is *not equal*. So, instructions after these instructions that may change condition codes should be divided into a new condition block even the predicated condition is the same.

4.6 Discussion

Our enhanced learning approach currently only supports user-level applications. The translation for full-system level applications is not supported because full-system translation is more complex with mechanisms such as system calls, interrupts and device I/O. These mechanisms make the learning and matching of rules more difficult. It is left in our future work.

ABIs and many instructions such as *indirect branches* are not supported either. For ABIs, the calling conventions, such as how parameters are passed and how many parameters are used, are difficult to be identified and translated by rules. For example, ARM use registers to pass parameters but no specific instructions are used. But in X86, the *push* instructions will be used for passing parameters. For indirect branches, DBT systems usually search the branch target address according to a branch table maintained at runtime, which is not available at compile time. It makes it impossible to translate by our learned rules.

5 Experimental Results

In this section, we evaluate our prototype and address the following research questions:

1. How much performance improvement can be obtained by our enhance learning scheme in which we relax the requirement of matching storage operands as described in Section 4?
2. Where does the performance improvement come from when we include the added constrained-equivalent translation rules?
3. What is the effect of relaxing the strict semantic equivalence requirement?
4. How much overhead will the dynamic analysis incur?

5.1 Experimental Setup

Our enhanced learning-based DBT prototype is implemented based on QEMU (version 2.6.0) which is the same as the original learning scheme. The guest ISA is ARM, and the host ISA is x86. The LLVM compiler (version 3.8.0) is used to generate binary code for guest/host ISAs. All binary codes are generated using the same optimization level -O2. The same version of source code and guest/host binary code are used for comparison. One machine with 3.33GHz Intel Core i7-980x with six cores (12 threads) and 16GB memory is set up exclusively for performance evaluation. The operating system is the 32-bit Ubuntu 14.04 with Linux 3.13 for both machines. We used an older version of the system because we need to compare our new approach with the original approach, which used the same older version of the system. Besides, our

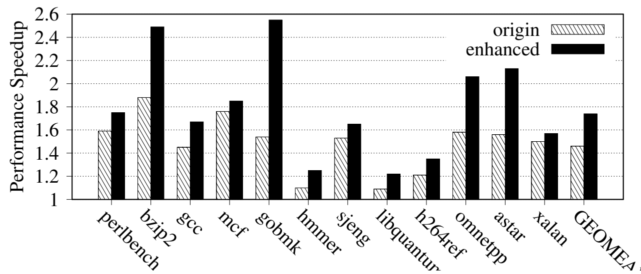


Figure 6: Performance comparison of the original and enhanced learning-based approaches.

experimental results are valid regardless of the system version used.

We use the 12 benchmarks in SPEC CPU INT 2006 with the reference inputs in our studies. To be as close to the real-world usage scenarios as possible, the performance of each benchmark is evaluated by the rules learned from other 11 benchmarks excluding evaluated benchmark itself. Each benchmark is run three times to reduce the random influence. The enhanced learning scheme described earlier is implemented in Python. The enhanced verification is implemented based on FuzzBALL [19], a symbolic execution engine. The tiny code generator (TCG) in QEMU is also enhanced to support the translation of predicated instructions as described in the last subsection, As mentioned earlier, to apply the constrained-equivalent translation rules, the dynamic analysis should be performed before such rules are applied.

5.2 Performance Results

Figure 6 shows the performance comparison of our enhanced learning-based scheme (marked as *enhanced*) and the original learning-based approach (marked as *origin*). The performance of QEMU without using any of the learning schemes is used as the baseline. Table 1 shows the MIPS of performance of original and enhanced approaches.

Using the *ref* input for all SPEC benchmarks, the quality of the translated host code is the major factor that impacts the overall performance. As shown in Figure 6, our enhanced learning scheme can achieve a performance improvement of up to 2.55X with an average of 1.74X compared to QEMU, which is a 1.19X improvement over the original learning approach on average.

By studying the learned translation rules and how they are applied using our enhanced learning approach, we have the following observations on how they impact the overall performance. First, constrained-equivalent translation rules can usually be applied quite successful. The modified guest registers/condition codes that have no matching modified host registers/condition codes will usually be modified quickly again. This means they are only used to hold temporary value as we expected. Hence, relaxing strict matching requirements

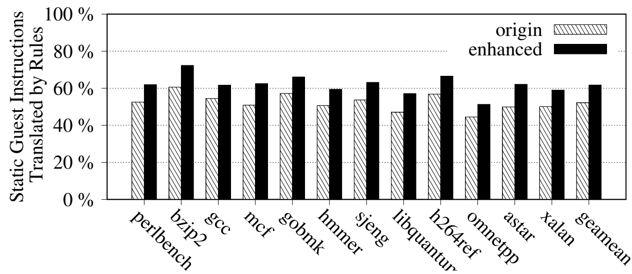
Table 1: MIPS of the original and enhanced learning-based approaches

Benchmarks	Original	enhanced
perlbench	221.63	250.15
bzip2	1211.32	1388.92
gcc	521.78	575.09
mcf	603.99	739.93
gobmk	372.18	616.31
hmmer	1448.56	1632.93
sjeng	474.71	485.23
libquantum	1469.59	1532.97
h264ref	189.99	215.69
omnetpp	195.28	284.74
astar	396.10	562.51
xalan	283.19	290.52
GEOMEAN	475.15	567.29

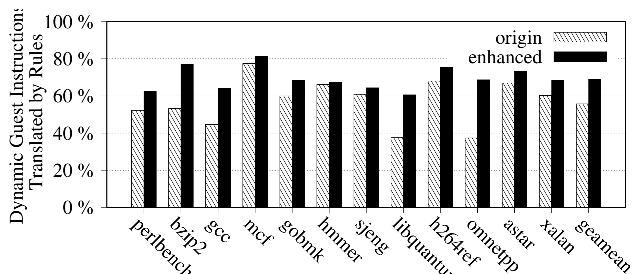
for storage operands can yield more translation rules, albeit constrained-equivalent rules, and can be applied quite effectively.

Second, by relaxing the equivalence constraints to allow constrained-equivalent translation rules that can include predicated and condition instructions greatly improve the overall performance. This is because typical DBTs such as QEMU usually use memory locations to emulate the condition codes. Such an approach will incur many additional memory operations to access and update those condition codes in memory and incur very high overhead. But the constrained-equivalent translation rules can take advantage of the host condition codes to emulate guest condition codes, which can significantly reduce such overheads.

Figure 7(a) and Figure 7(b) show the static and dynamic coverage of the guest binaries using the origin and our enhanced learning-based schemes, respectively. The *"Coverage"* here is defined as the percentage of guest instructions that can be translated by the learned rules. So the *"static"* coverage is the percentage of static code translated by learned rules and *"dynamic"* coverage here is the percentage of *"executed"* guest instructions translated by learned rules. Compare to the original learning-based scheme, our enhanced learning scheme can improve the static coverage from 52.2% to 61.8%, and the dynamic coverage from 55.7% to 69.1% on average. It is worth noting that *gcc* and *libquantum* have a much higher dynamic coverage improvement than others, but do not get an expected higher performance improvement. Conversely, *gobmk* attains a high performance improvement but not as much coverage improvement. The reason is that many high-quality rules are applied when translating *gobmk*, but in *gcc* and *libquantum*, the applied rules can only attain moderate improvement. This seems to indicate that the coverage improvement does not translate directly to the overall performance improvement, but could be an important secondary effect.

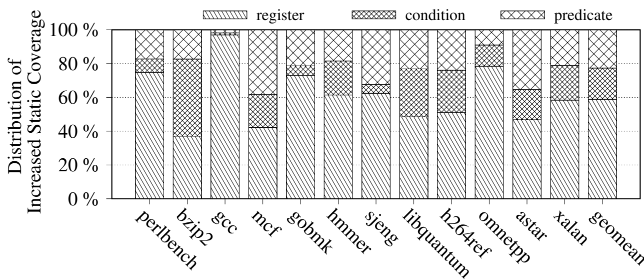


(a) Static coverage

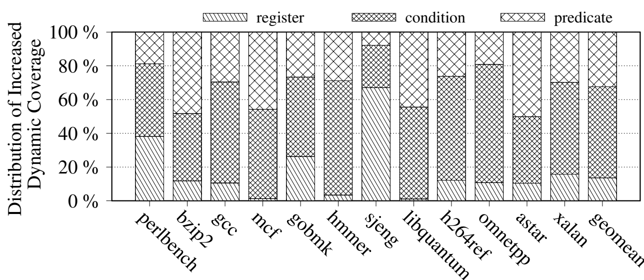


(b) Dynamic coverage

Figure 7: Static and dynamic coverage of translation rules.



(a) Distribution of increased static coverage



(b) Distribution of increased dynamic coverage

Figure 8: Distribution of the improved rule coverage.

To address the question of "where does the performance improvement come from when we include the added constrained-equivalent translation rules?", we analyze the coverage of the added translation rules when they are applied in each program. The results are shown in Figure 8. As described in Section 4, there are three major components in our relaxed equivalence constraints, i.e. we remove the strict requirement of exact matching. They are (1) register operands (marked as *register*), (2) condition-code operands (marked as *condition*), and (3) predicated-related instructions (marked as *predicate*). We did not include memory operands because they require more complicated data dependence analysis when they are applied.

A very interesting observation is that, among the added constrained-equivalent translation rules (their increased coverage is shown in Figure 7), the register-related constrained-equivalent translation rules constitute 58.83% of the static instructions on average. However, they only constitute 13.58% of added dynamic coverage. But the dynamic coverage of condition-code related rules is increased to 54.02% on average, while their static coverage is only 18.52%. This is because the condition codes are usually associated with the bound check, such as at the end of a loop. So, these instructions will be executed more frequently than others in their dynamic coverage.

To study the quality/efficiency of translated rules, Figure 9 shows the percentages of the reduced host instructions. On average, our enhanced learning scheme can reduce 11.28% of the total dynamic host instructions compared to the original learning scheme. We observe that the reduction in the host

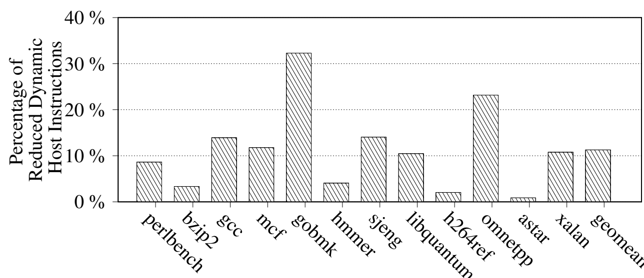


Figure 9: Reduction in dynamic host instruction counts by enhanced learning and translation.

instructions of *gobmk* is higher than 30%, and in *omnetpp*, it is higher than 20%. However, in *gcc* and *libquantum*, the reduction is only about 10%. This also confirm our observation that rules applied in *gobmk* and *omentpp* translation have a higher quality, i.e. fewer host instructions in those translation rules, than rules in *gcc* and *libquantum*. But, we also notice that *bzip2* and *astar* attain a high performance improvement but only a moderate number of host instructions are reduced. One probable explanation is that even though they may have similar dynamic host instruction counts, more efficient and architecture-specific host instructions may have been used.

5.3 Learning Results

We further study the effect of our proposed relaxed learning scheme in other related aspects. The first is about the "yield" obtained during the learning phase, which shows how

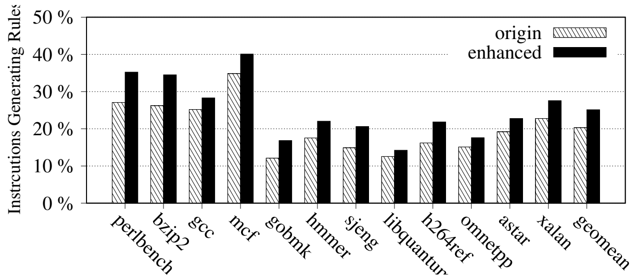


Figure 10: The yields of rule learning.

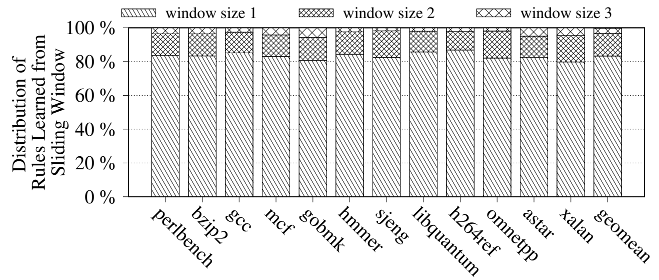


Figure 11: Rules distribution of sliding windows.

many translation rules can be harvested among the candidate rules during the learning phase. The other aspects include the effect of using the sliding window, and the distribution of the constrained-equivalent rules learned based on our relaxed equivalence constraints, i.e. relaxing the strict matching requirement on three storage operand types described in Section 4.

Figure 10 shows the yield obtained during the learning phase using the original learning scheme (marked as *origin*) and our enhanced learning scheme (marked as *enhanced*). The learning yield is increased from 20.3%to 25.1%. Even though the improvement in learning yield is moderate as we only made moderate relaxation on the semantic equivalence requirements, but as the obtained performance results show the quality of these rules is quite high. The moderate yield improvement also shows that there is a high potential for more high-quality rules to be learned and harvested. Another interesting question is that if a significant number of more source programs is used in the training phase, even with a low yield, how much more rules can be learned, and how much more performance improvement can be achieved by applying those added rules. These questions are beyond the scope of this paper.

Figure 11 shows the distribution of the translation rules learned using a flexible sliding window. We only show the data for a window size of up to three source statements because significantly fewer rules can be learned beyond 3 source statements. As the result shows, 13.16% of new rules can be learned from a window size of 2 and 3 source statements. The rules learned from window size 3 and beyond are less than 3.31%. So, a larger learning window is not necessary.

Figure 12 shows the distribution of the rules we learned using our enhanced learning scheme. On average, 16.84% of the learned rules are register-related (marked as *register*), while 8.16% are condition-code related (marked as *condition*) rules. We find that many constrained-equivalent rules related to local registers are used to load values from the memory before some computation, and are stored back to the memory after the computation. This is because RISCs are primarily "load/store" architectures, i.e. values in memory must be loaded into registers before computation and stored back to memory when the computation is completed. So many

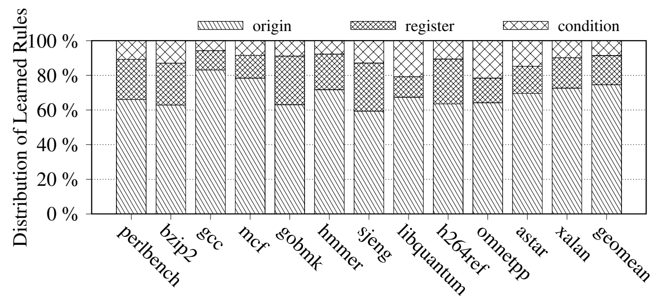


Figure 12: Distribution of rules learned by enhanced learning.

temporary/local registers are used. Another observation is that the amount of register reuse is minimal on RISCs, only 3.57% in total. So, only in rare situations, the compiler will use multiple registers instead of only one register. Such behavior is reflected in the use of the learned rules in the application phase.

5.4 Performance Overhead of Online Analysis

As the lightweight dynamic analysis is needed in the application of the constrained-equivalent translation rules, its runtime overhead needs to be evaluated. Figure 13 shows such runtime overhead with and without dynamic analysis. To measure such overheads, we collected the performance data with original approach and compare them with those with only the online analysis but without applying the constrained-equivalent rules. As Figure 13 shows, the dynamic analysis will introduce very little overhead, which is less than 1% on average. The low overhead is due to two main reasons. First, the dynamic analysis typically only needs to check a few registers and condition codes. And the percentage of the rules that requires dynamic analysis is not very high. Second, the relaxed register and condition-code operands are usually updated very quickly, so only a very small number of instructions need to be analyzed in practice. Both factors greatly reduce the analysis overhead.

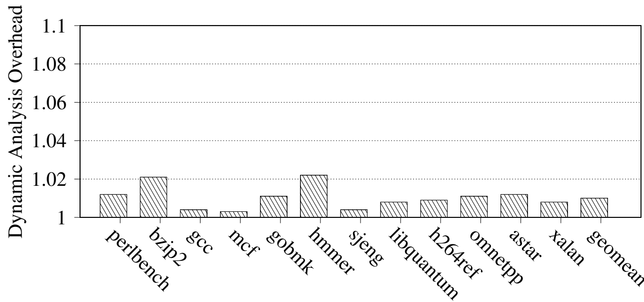


Figure 13: The performance overhead of dynamic analysis.

6 Related Work

To improve the efficiency of the translated host binaries, many manual optimization techniques have been proposed. For example, some try to efficiently translate guest single-instruction-multiple-data (SIMD) instructions [7, 10, 16]. Another work proposes to leverage host hardware features to apply post-optimization to host binary code after the translation [30]. Some recent work also proposes to optimize dynamically-generated guest binary code [8]. Different from those approaches, most of which rely on manually constructed translation rules, our enhanced learning-based approach proposed in this paper can automatically learn binary translation rules.

Previous work in [1] also proposes to use peephole superoptimizer to generate binary translation rules for static binary translators. For each potential guest code sequence, an exhaustive search is employed to explore all possible sequences of host instructions to examine their equivalence. However, it takes a very long time to collect sufficient translation rules, i.e., could be up to one week as mentioned in the paper. Moreover, due to the exponential increase in the number of possible instruction sequences, this approach can only generate translation rules with a guest code sequence of up to 3 instructions. This can significantly limit the quality of the generated translation rules because many high-quality translation rules have more than 3 guest instructions.

Although the learning-based approach was originally proposed in [23], our enhanced learning-based approach differs from the original approach in a significant way. Our proposed enhanced approach allows relaxation of semantic equivalence, thus can learn constrained-equivalent translation rules while the original approach simply discards them. These relaxed translation rules can improve the total coverage of the guest binaries and improve the yield of rule generation. More importantly, these constrained-equivalent translation rules can further improve the performance of the translated host binary code.

Another DBT system, HQEMU [9], which is also based on QEMU, translates guest binary code into LLVM intermediate representation (IR) and then leverages LLVM JIT to generate more optimized binary code. However, the overhead introduced by the LLVM optimization can offset the benefit gained from the optimized host binary code, especially for short-running guest binaries. Moreover, due to the lack of source-level information in the LLVM IR translated from the guest binary code, e.g., type information, it is quite challenging to take full advantage of the LLVM optimization. In contrast, the translation overhead for applying the learned translation rules are much smaller, and no additional information is required to apply the learned rules.

There has been a lot of research to improve the performance of the DBT system itself [3–5, 11, 12, 24, 25]. These methods can typically be used in conjunction with our approach to further improve their performance.

7 Conclusion

As one of the core enabling technologies, DBT has been extensively used in many important applications. To improve the efficiency of DBT systems, this paper proposes an enhanced learning-based approach, which can automatically learn optimized binary translation rules. The learned translation rules can then be applied to a DBT system to generate more efficient host binary code. Compared to the original learning approach, our enhanced learning-based approach relaxes the semantic equivalence requirements to allow more efficient constrained-equivalent translation rules. We redesign the original learning process and the verification engine to accommodate such constrained equivalence. Moreover, to preserve the correct semantics of the translated code when such constrained-equivalent translation rules are applied, a lightweight online analysis is employed in the enhanced DBT system to check the constraining conditions. The constrained-equivalent translation rules are applied only when the constraining conditions are satisfied. We have implemented the proposed approach in a prototype and extended a widely-used DBT system, i.e., QEMU, to accept such enhanced translation rules through learning.

Experimental results on SPEC CINT2006 show that the proposed approach can improve the dynamic coverage of the translation from 55.7% to 69.1% and the static coverage from 52.2% to 61.8%, compared to the original approach. Moreover, up to 1.65X performance speedup with an average of 1.19X are observed.

Acknowledgments

We are very grateful to our shepherd, Edouard Bugnion, and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Natural Science Foundation of China (No. 61672160), Shanghai Municipal Science and Technology Major Project (No.2018SHZDZX01) and ZJLab, Shanghai Sci-

ence and Technology Development Funds (17511102200) and the National Science Foundation under the grant number CNS-1514444.

References

[1] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association.

[2] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[3] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 117–128, New York, NY, USA, 2014. ACM.

[4] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 210–220, Piscataway, NJ, USA, 2017. IEEE Press.

[5] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on arm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 333–346, New York, NY, USA, 2017. ACM.

[6] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 135–146, New York, NY, USA, 2012. ACM.

[7] Sheng-Yu Fu, Ding-Yong Hong, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. Dynamic translation of structured loads/stores and register mapping for architectures with simd extensions. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2017, pages 31–40, New York, NY, USA, 2017. ACM.

[8] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society.

[9] 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. ACM, 2012.

[10] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. Improving simd parallelism via dynamic binary translation. *ACM Trans. Embed. Comput. Syst.*, 17(3):61:1–61:27, February 2018.

[11] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 23–32, New York, NY, USA, 2013. ACM.

[12] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 1–12, New York, NY, USA, 2013. ACM.

[13] Piyus Kedia and Sorav Bansal. Fast Dynamic Binary Translation for the Kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, New York, NY, USA, 2013. ACM.

[14] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented Reverse Engineering of Functional Components from x86 Binaries. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1128–1139, New York, NY, USA, 2014. ACM.

[15] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[16] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. Optimizing dynamic binary translation for simd instructions. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 412–425, New York, NY, USA, 2018. ACM.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[19] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM.

[20] Aashish Mittal, Dushyant Bansal, Sorav Bansal, and Varun Sethi. Efficient Virtualization on Embedded Power Architecture®Platforms. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 445–458, New York, NY, USA, 2013. ACM.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[22] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[23] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. ACM.

[24] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. Improving dynamically-generated code performance on dynamic binary translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pages 17–30, New York, NY, USA, 2018. ACM.

[25] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 591–603, Berkeley, CA, USA, 2016. USENIX Association.

[26] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 319–331, New York, NY, USA, 2017. ACM.

[27] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM.

[28] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. ACM.

[29] W. Zhang, X. Ji, Y. Lu, H. Wang, H. Chen, and P. Yew. Prophet: A parallel instruction-oriented many-core simulator. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2939–2952, Oct 2017.

[30] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. Hermes: A fast cross-isa binary translator with post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society.

Transactuations: Where Transactions Meet the Physical World

Aritra Sengupta
Samsung Research

Tanakorn Leesatapornwongsa [*]
Samsung Research

Masoud Saeida Ardekani [†]
Samsung Research

Cesar A. Stuardo [‡]
University of Chicago

Abstract

A large class of IoT applications read sensors, execute application logic, and actuate actuators. However, the lack of high-level programming abstractions compromises correctness especially in presence of failures and unwanted interleaving between applications. A key problem arises when operations on IoT devices or the application itself fails, which leads to inconsistencies between the physical state and application state, breaking application semantics and causing undesired consequences. Transactions are a well-established abstraction for correctness, but assume properties that are absent in an IoT context. In this paper, we study one such environment, smart home, and establish inconsistencies manifesting out of failures. We propose an abstraction called *transactuation* that empowers developers to build reliable applications. Our runtime, *Relacs*, implements the abstraction atop a real smart-home platform. We evaluate programmability, performance, and effectiveness of transactuations to demonstrate its potential as a powerful abstraction and execution model.

1 Introduction

Building reliable IoT applications that interact with the physical world on top of existing solutions is difficult. Current IoT solutions (e.g., Smartthings [14] and OpenHAB[12]) provide simple abstractions that allow developers to easily read sensors and actuate actuators. However, they lack high-level abstractions for writing reliable and fault-tolerant applications that can tolerate different types of failures that might happen. Therefore, application programmers need to implement tedious and error-prone code for not only handling all kinds of failures happening in the physical world, but also to guarantee consistency between operations on application states (called soft states hereafter) and states of IoT devices (called hard states). For instance, an actuation to turn on an alarm might fail while the alarm state in an application might have been set to true.

The use of serverless functions as a de facto platform for running IoT applications has exacerbated the reliability issues of these applications even further. This is because serverless computing infrastructure can terminate running applications at any point [2]. This again leaves incomplete operations on some hard states (e.g., lock all doors) inconsistent with an operation on soft state inside the application (e.g., set the home state to `safe` after all doors are locked).

Transactions seem like the right mechanism for addressing the above issues. Interestingly though, a transactional abstraction cannot fix these issues because of intrinsic properties of IoT devices (and their associated hard states). A transactional abstraction is ideal for ensuring isolation and all-or-nothing guarantees among soft states. Moreover, a transactional system can easily rollback soft states without other transactions or users noticing effects of a rolled back transaction. However, rolling back a hard state has consequences. The state might have already been observed by a user and rolling it back may be undesirable. Or even worse, some states cannot be rolled back (e.g., undoing actuation of a water dispenser).

This paper proposes an abstraction called *transactuation*. Transactuations hide the complexity of handling various failures and allow developers to easily maintain soft states to be consistent with respect to reads and writes to hard states – states of sensors and actuators. Objectively, transactuations allow a developer to specify dependencies among operations on soft and hard states along with a *sensing/actuating policy* which specifies the conditions under which soft states can commit despite failures.

We provide a runtime system called *Relacs* that implements the abstraction for the smart home environment. Relacs transforms an application into a serverless function, and reliably executes the application in the cloud while enforcing transactuation specific semantics. We note that while the focus of this paper is on smart homes, the transactuation abstraction is not particularly specific to smart homes, and can be applied to other IoT environments as well.

[*] Work done at Samsung Research America. Now at Microsoft Research.
[†] Work done at Samsung Research America. Now at Uber Technologies.
[‡] Work done at Samsung Research America.

Concretely, this paper has the following contributions:

1. *Study of smart-home applications.* Using static analysis, we conduct a comprehensive study of smart-home applications written for two popular platforms [12, 14] and identify drawbacks of existing platforms in writing reliable and fault-tolerant applications (Section 3).

2. *Transactuations.* We present our abstraction that allows developers to simply write reliable IoT applications. Transactuations preserve the dependencies between operations on hard states and soft states, which when broken, break application semantics (Section 4).

3. *Relacs.* Our runtime, Relacs, enforces a serializable execution of transactuations without rolling back hard states (i.e., states of actuators) while enforcing the specified sensing and actuating policies (Section 4 and Section 5).

4. *Evaluation.* We evaluate representative smart-home applications to reveal the correctness issues due to lack of appropriate abstractions. Our evaluation further demonstrates that (a) Transactuations are an effective high-level abstraction for building reliable IoT applications and reduce lines of code significantly compared to manually handling failures. (b) Relacs guarantees reliable execution of transactuations while imposing reasonable overheads over a baseline that does not provide consistency between operations on hard states and soft states (Section 6).

2 Background & Model

In this section, we first review existing smart-home platforms and their programming models. We focus on smart homes as a case study of class of IoT environments that deal with real world state since many smart home applications and platforms are publicly available. We then discuss different types of failures that occur in IoT environments.

2.1 Smart-home Platforms

To setup a smart home, a user installs centralized gateways, called smart-home hubs or simply hubs, to connect in-home devices (e.g., light bulbs, outlet strip, and motion sensor) that typically communicate through low-energy wireless protocols (e.g., Zigbee [17], ZWave [16], and Bluetooth Low Energy [4]). The user then installs smart-home applications to create her desired home automation. For instance, to turn on a balcony light when motion is detected outside.

Currently, cloud-centric smart-home solutions (e.g., Smartthings [14]) are the most widely used architecture [28]. In this model, a hub is only responsible for collecting device events, and forwarding them to the cloud, where applications run. The applications running in the cloud then process events and send actuation commands back to the hub, which forwards the commands to corresponding devices. An alternative architecture is to run applications inside hubs. OpenHAB [12] follows this hub-centric approach.

```
1   preferences {
2     input(sensor, "capa.co2", req:true)
3     input(switches, "capa.switch", multi:true)
4     input(level, "number", req:true)
5   }
6   def initialize() {
7     state.active = false;
8     subscribe(sensor, "co2", handleLevel)
9   }
10  def handleLevel(evt) {
11    def co2 = sensor.currentValue("co2");
12    if(co2 >= level && !state.active) {
13      switches.each { it.on(); }
14      state.active = true;
15    } else if(co2 < level && state.active) {
16      switches.each { it.off(); }
17      state.active = false;
18    }
19  }
```

Listing 1: CO_2 vent application that turns exhaust fans on when CO_2 level is high and turns off otherwise.

2.2 Programming Model

In most smart-home platforms, an application is written in a trigger-action programming model [45] where an application comprises event handlers. Handlers can subscribe to changes in sensor/actuator states, updates to shared states, or timer-based events. Handlers can issue the following operations:

- Hard read: reading sensor/actuator values.
- Hard write: sending actuation commands to actuators.
- Soft read: reading application states from shared storage.
- Soft write: writing application states to shared storage.

In the remainder of this section, and for simplicity, we solely detail SmartThings [14] programming model. Yet, we note that other platforms have very similar constructs.

SmartThings uses capabilities, attributes, and commands to manage devices. Each device has one or more capabilities, and each capability has one or more associated attributes and commands. For example, a smart light bulb has two capabilities, switch and color. The switch capability allows an application to control the bulb status via `on`/`off` commands. The color capability has three attributes, color, hue and saturation that can be controlled via `setColor`, `setHue`, and `setSaturation` commands.

Listing 1 shows a SmartThings application, named CO_2 vent, written in the Groovy language [5]. It reads CO_2 level from sensors, and turns on an exhaust fan if the level is high. Similarly, it turns off an exhaust fan if the level is low. A developer first declares mapping of variable names to capabilities in the preference section (lines 1-5). Consequently, a variable is mapped to an array of devices with the same capability. For example, variable `switches` (line 3) gets mapped to an array of exhaust fans having the switch capability.

A developer then subscribes event handlers to value changes of some capabilities or timer schedules. In line 8, she subscribes an event handler called `handleLevel` to co2 capability. Observe that inside the handler, she can perform

hard read on sensor data (`sensor.currentValue()` in line 11) and soft read on shared states (reading `state.active` in line 12 and 15). Also, the developer can issue hard writes to list of actuators (line 13 and 16). She can also perform soft writes to application states (assignments to `state.active` in line 14 and 17).

2.3 Failures in IoT Environments

Previous work [20, 33] have shown a variety of failures in IoT environments. For instance, hubs can fail due to plug disconnection, hardware failure, and driver crash. IoT devices can fail due to battery drainage, plug disconnection, and failure in a sensor subsystem. Additionally, network loss occurs due to RF interference, concrete slab flooring and copper siding. These failures lead to permanent or intermittent unavailability of devices in an IoT environment.

Although, these failures are common, existing platforms do not provide a simple way to detect and handle them. A failed hard read can produce a null or stale value that a developer needs to handle or explicitly validate its timestamp (freshness). Detecting a failed hard write is even more difficult due to the asynchronous nature of IoT programming model. For instance, a developer needs to subscribe to an event triggered by a hard write, and periodically check if the event is fired. As shown in other systems [30, 35, 37], inserting failure detection and handling code for asynchronous environments is challenging and error prone. Moreover, due to inherent event-driven concurrency in applications, it is notoriously difficult to prevent interleaving and concurrency-related bugs in IoT platforms [40].

3 Problem Study

Existing smart-home solutions do not guarantee any consistency between soft reads/writes (i.e., reads/writes from/to shared storage) and reads/writes to hard states (i.e., sensor reads and actuation commands sent to actuators) in case of failures. Application developers need to carry the burden and ensure the correctness of an application when a failure occurs.

In this section, we present a systematic study of open source smart-home applications, using static analysis, in order to unearth various inconsistencies, that surface under failure, between operations on soft and hard states.

3.1 Inconsistency

Listing 2 shows a simplified code excerpt from a smart security application. This application associates a soft state named `alarmActive` with the status of an alarm. If the application detects an intruder when the alarm is not active, it activates the alarm and sets `alarmActive` to `true`. However, an inconsistency arises if the alarm is not activated properly. For example, RF interference may cause an actuation command to be lost. This problem is so common that some brands (e.g., Fortrezz [15]) give warnings regarding RF interference, and explicitly ask consumers to not use the alarm in life supporting situations. Observe that even though `alarmActive` is set, the states of the physical world and application have diverged. Further, if the sensors detect the intruder again, the application will not retry to activate the alarm because as per the application's state the alarm is ringing. Clearly, the developer does not anticipate such a failure, and this divergence is irreversible without manual intervention. Such inconsistencies cause changes in application semantics and compromise correctness, and may severely affect smart-home users.

```
1  def intruderMotion(evt) {
2    ...
3    if (isIntruder(evt) && !state.
         alarmActive) {
4      alarm.strobe();
5      state.alarmActive = true;
6    }
7    ...
8  }
```

Listing 2: A simplified code excerpt from Smart Security application that detects an intruder using sensors, and activates an alarm if it has not been activated previously.

Moreover, stale hard reads may also break correctness of an application. For example, recent CO_2 level events might never get delivered to the CO_2 vent application in Listing 1. By reading a stale CO_2 level, the application may incorrectly turn off the exhaust fans.

Besides device failures, similar issues arise if an application crashes. For instance, an inconsistency arises if the smart security application fails between sending a command to set the alarm (line 4) and setting the active state to true (line 5).

Finally, applications may modify shared soft and hard states concurrently [40] which can cause canonical interleaving based inconsistencies [39].

As an example, the following quote from a disgruntled SmartThings customer [9] who got robbed during his vacation shows the impact of the inconsistency problem: "*More importantly, we were robbed when we were out on vacation. ... The logs show the motion of the robbers, but it never sounded the alarm ... I no longer trust it to do what it is supposed to do when it is supposed to do.*"

3.2 Dependency

In the previous section, we showed connections between hard states and soft states that are potential sources of inconsistencies due to hard read/write failure. We call these connections between two operations on hard states or two operations between soft and hard states that are semantically associated, a *dependency*. By identifying dependencies in an application, we can study the effects of failures on its correctness.

In order to systematically analyze smart-home applications, and understand how failures can affect them, we categorize

dependencies into four classes, using the following notations: we represent a hard read to device D as HR_D, and denote a hard write to device D with value V as $HW_D(V)$. A soft read from application state X is denoted as SR_X, and a soft write to state X with value V is represented as $SW_X(V)$.

1. $HR_D \rightarrow HW_{D'}(V)$: a dependency in this category captures the effect of a failure in a HR_D. The read might fail to return any value if device D is unavailable, or it might return a stale value. In either case, it implies that the application may exercise the dependency incorrectly, thus breaking its semantic. Such a dependency in an application can be because of a control dependence [27] or a data dependence.

If the dependency is a control-dependence [27], the value of HR_D controls the execution of $HW_{D'}(V)$. For example, in Listing 1, the dependency between lines 12 and 13, and also between lines 15 and 16 are control dependences. The hard read in line 11 flows into the control statements in lines 12 and 15. Therefore, a stale read at line 12 might incorrectly switch off the exhaust fans, and update the soft state even though the CO_2 levels are unsafe. A read value can also flow into a hard write via data dependencies. For example:
`a =` HR_{D_1}`; c = foo(a, b);` $HW_{D_2}(c)$.

2. $HR_D \rightarrow SW_X(V)$: this dependency affects the execution of a soft write. Analogous to $HR_D \rightarrow HW_{D'}(V)$, this results in a missing soft write or an incorrect soft write, because of control and data dependences. In turn, the incorrect soft write leads to unexpected program behavior when the state is read elsewhere. In our running example, this dependency exists between lines 12 and 14, and also lines 15 and 17.

3. $HW_D(V) \rightarrow SW_X(V')$: a hard write to soft write dependency is more subtle since $SW_X(V')$ is not a control or a data dependence on a $HW_D(V)$. Nevertheless, we observe that semantically tying a soft state with a hard state — meaning the soft state is an indicator of the hard state — is a common practice in many smart-home applications. Developers use this technique mainly to save battery: by associating a soft state with an actuation, developers can use the soft state elsewhere in the code instead of reading hard states.

For example, in the CO_2 Vent application, the developer implicitly creates a $HW_{switches}(ON) \rightarrow SW_{active}(true)$ dependency between lines 13 and 14, and also between lines 16 and 17. Thus, a failure in turning on `switches`, even if temporary, leaves a permanent inconsistency. Any subsequent change in the CO_2 level, even above the `level`, precludes turning on the exhaust fans.

To find a $HW \rightarrow SW$ dependency in the code, we compute the postdominance relation [23]: a code point b postdominates a code point a, if b is executed on every path from a to the end of the analyzed entity, which in our case, is an event handler. After computing postdominance instances, we manually look at all instances to confirm if the pair is semantically tied. Accordingly, we infer a case for semantic error if the soft state is read elsewhere in the application.

4. $SW_X(V) \rightarrow HW_D(V')$: this dependency has the same semantic effect as $HW_D(V) \rightarrow SW_X(V')$.

Note that all dependencies with soft reads (i.e., $SR_X \rightarrow *$), are not directly related to device failures. However, we still statically compute all such control and data dependences as an incorrect soft read can produce unintended behavior. Concretely, a soft read can be on a state determined by an incorrect, inconsistent, or missing soft write originating from the dependencies described above.

3.3 Analysis and Findings

We statically analyzed 147 SmartThings applications [19] and 35 OpenHAB applications chosen from IoTBench [10] by adding phases to the Groovy compiler. The AST visitors, `GroovyClassVisitor` [8], allow us to build a call graph per entry point and an intermediate representation (IR) amenable to data and control-flow analysis.

We analyzed the applications using inter-procedural data and control-flow analysis to understand the dependencies and their implications. Our analysis yields two key benefits: (i) understand the implications and the extent of failures on a large set of smart-home applications, and (ii) mitigate or eliminate the problems with our programming abstraction, called transactuation.

On average, the studied applications have three triggers, and manage a diverse set of devices (4–5 capabilities). In order to get a holistic view of the home state, on average, the applications perform three hard reads. They also perform between seven to nine hard writes on average. This shows that many of these applications try to provide automation among a set of devices (e.g., turning on restroom light, preparing coffee, and playing music, when a user wakes up), instead of managing a single device. Additionally, our analysis revealed that developers regularly use soft states to share states not only among handlers, but also among different applications. These results indicate that smart-home applications are fairly complex, and their behavior could be complicated through the use of handlers triggered by events that read/write both hard and soft states.

More specifically, we observed that, on average, applications have 3–10 instances of $HR \rightarrow HW$, 1–2 instances of $HR \rightarrow SW$ and 1–2 instances of $HW \rightarrow SW$ dependencies. We inspected these dependencies to find their potential implications on systems lacking appropriate abstractions to capture failures. We categorized the implications as follows: (i) missing actuation, (ii) wrong actuation, (iii) inconsistent soft state, (iv) missing notification, and (v) wrong notification . These implications can lead to unwanted outcomes, some of which have serious consequences such as security threats, health hazards, and missing critical alerts, e.g., a fire alarm not rung. They may also cause inconveniences, e.g., erroneous automation, incorrect notifications, sirens not turned off. Out of all 182 applications, our analysis unearthed 67 SmartThings and 32 OpenHAB applications, that have unintended effects. Due

Application	Type	Consequence	Dependency	Correction/Mitigation
Smart Humidifer (ST)	Automation	Wrong status flag causes humidifier to never be turned on/off. Incorrect notification.	*HW*→ *SW* *SR*→ *	Correct status flag to retry turning humidifier on/off later. Notify glitch to user.
Thermostat Auto Off (ST)	Energy	Wrong status flag causes thermostat to never be turned on/off.	*HW*→ *SW*	Correct status flag to retry turning thermostat on/off later.
CO2 Vent (ST)	Safety	Wrong status flag causes exhaust fans to never be turned on/off.	*HW*→ *SW* *SR*→ *	Correct status flag to retry turning exhaust fans on/off later.
Elder Care (ST)	Safety	Missing elder inactivity notification	*HR*→ *HW*	Notify glitch to user.
Smart Care (ST)	Safety	Alarm not armed. Missing notification.	Bad interleaving *HW*→ *SW*	Notify glitch to user.
Alarm (OH)	Security	Wrong status flag causes sirens to never be turned off.	*HR*→ *SW* *SR*→ *HW*	Correct status flag to retry turning sirens off.
Fire Detection (OH)	Security	Wrong status flag causes fire alarm to never ring	*HW*→ *SW* *SR*→ *	Correct status flag to retry ringing alarm later.
Forgiving Security (ST)	Security	Alarm does not ring. Incorrect notification.	*HW*→ *SW* *SR*→ *	Notify glitch to user.
Lock It When I Leave (ST)	Security	Door not locked but home vacant.	*HR*→ *HW*	Notify user to lock manually.

Table 1: Critical undesirable consequences in smart-home applications if failures are not handled and how developers can correct or mitigate the problems. ST and OH are abbreviations for SmartThings and OpenHAB, respectively.

to space constraint, we only show a subset of them with unintended semantics and potential fixes in Table 1.

To address these implications, a developer needs to preserve the semantic invariants of the dependencies to avoid discrepancy between the physical and application realms. One key trait of these applications is that their semantics tolerate different numbers of failed hard reads and writes. For example, for *HR*→*HW* in the application that computes average humidity level and reacts accordingly, even if some hard reads are stale based on their timestamp (i.e., some humidity sensors fail), the application can proceed with correct semantics as long as some sensors function properly. On the other hand, for *HW*→*SW* in the application that locks all doors and set the home state to `safe`, the developer needs to ensure that the home state is not set, even if only one door fails to be locked. To summarize, the following two key aspects are missing in existing IoT abstractions: 1. identifying the inherent connection between application semantics and *number* of failed operations, and 2. recomputing application states to preserve invariants under failed hard reads/writes.

4 Transactuations

To address the issues discussed in the previous section, we introduce a new abstraction called *transactuation* that allows a developer to build a reliable smart-home application. Transactuations provide the following two guarantees: (1) preserve dependencies between reads/writes to hard states and soft writes (i.e., *HR*→*SW* and *HW*→*SW*) even in cases of hardware and communication failures. (2) ensure isolation among transactuations that execute concurrently.

The concept of transactuations is very similar to database transactions. Yet, due to the intrinsic nature of physical world, it is impossible to ensure similar transactional guarantees. We note that transactuations are not meant to replace transactions completely. Instead, they are designed to address a similar problem in a cyber-physical environment which inherently prevents us from making strong assumptions. Precisely, transactuations and transactions differ as follows:

1. *Atomic durability:* atomic durability [36] guarantees that either all updates inside a transaction eventually become durable, or none of them becomes durable. Since IoT devices can neither be locked nor rolled back (e.g., in case of some failures), transactuation cannot guarantee atomic durability of hard writes. More specifically, unlike a transaction, a transactuation only guarantees atomic durability of soft writes but not hard writes inside it. Thus, if a hard write fails, a transactuation still commits by forcing its soft states to be consistent with its hard states, as per developer specified policies (see Section 4.1).

2. *Isolation & Atomic visibility:* strong isolation models (e.g., serializability or snapshot isolation) requires a transaction to read a consistent snapshot of a system (e.g., the last committed state) and precludes a use of partially committed states. A transactuation executes on the latest known consistent snapshot of the physical world, in isolation from other concurrent transactuations. However, two concurrent transactuations can execute on different snapshots of the physical world in absence of any committing transactuation. Additionally, (internal) atomic visibility ensures that effects of all updates in a transaction become visible to another transaction atomically [36]. Transactuations are also capable of guaranteeing internal atomic visibility: effects of a transactuation become atomically visible to other transactuation. However, in a smart home domain, consumers will unavoidably observe the effect of a hard write operation the moment it gets

executed in an actuator. Thus, it is impossible to provide *external* atomic visibility. For instance, one cannot expect that a smart-home user to observe all door locks become locked instantaneously.
Transactuations, further add to the definition of consistency based on consistency between hard reads/writes and soft writes. Transactuations preserve two invariants as follows:
(D1) A transactuation guarantees that if it executes, the staleness of its hard reads is bounded, as per the developer specified tolerance. A developer leverages this invariant to ensure inconsistencies arising out of breaking *HR*→ * dependencies are detected, and appropriate actions are taken.
(D2) If writes to soft states are committed, it implies that sufficient number of hard writes as per developer specification have successfully executed. A developer leverages this invariant to enforce consistency of *HW*→ *SW* dependencies.

4.1 Abstraction & API

Transactuations contain three pieces of logic which a developer writes as lambda expressions. A lambda expression is a function that can be passed as an argument to another function [1, 7, 11]. In the rest of this paper, we refer to these lambda expressions as lambdas. A transactuation can have the following three lambdas: perform lambda, onSuccess lambda, and onFailure lambda.

perform lambda. A perform lambda contains the core logic of a transactuation. Inside a perform lambda, a developer can perform hard writes (`actuate(Device, Value)`), soft reads (`read(State)`), and soft writes (`write(State, Value)`) as shown in Listing 3.

To assign a perform lambda to a certain transactuation, a developer calls the `perform()` method and passes the lambda as an argument as shown in lines 5–15 of Listing 3. The method signature is `perform(performLambda, [sensorList, timeWindow, sensingPolicy], [actuatingPolicy])`.

A developer cannot explicitly issue a hard read inside a perform lambda. Instead, she has to specify a list of required hard states as an argument (i.e., `sensorList`) to `perform()` method. The required hard states are read before perform lambda is executed, and a list of available hard states are accessible as key-value pairs to perform lambda, using `sensors` parameter of a perform lambda (line 5). Disallowing explicit hard reads inside a transactuation prevents reading stale or null sensor values, which can break application semantics.

To preserve consistency between hard reads and soft writes in case of a sensor unavailability, a developer can use a time window along with a sensing policy. The time window specifies that the sensor list must be validated such that, after validation, the list of available sensors includes those that have received events close in time. Specifically, a time window defines the duration when the transactuation triggering event and read hard states remain valid. For instance, a window of

```
1  function handler(evt) {
2    let tx = Transactuation(evt);
3    // executes if all CO2 sensors received
4    // events in past 5s w.r.t. triggering event
5    tx.perform(func(sensors){
6      let co2 = sensors['co2'];
7      let active = read('active');
8      if (co2 >= threshold && !active) {
9    //if all fans can be on, set active to true
10       actuateAll('fans', 'on');
11       write('active', true);
12     } else if (co2 < threshold && active) {
13       ...
14     }
15   }, ['co2'], 5, 'all', 'all');
16   // executes if both policies are met
17   tx.onSuccess(func(evt) {
18     let txs = Transactuation(evt);
19     txs.perform({
20       actuate('msg', 'CO2 is high');
21     }, 'none', 'none');
22     txs.execute();
23   });
24   // executes if either one policy is not met
25   tx.onFailure(func(evt) {
26     let txf = Transactuation(evt);
27     ...
28   });
29   tx.execute(); }
```

Listing 3: CO2 Vent written with transactuation. The code presented here is in synchronous style but our implementation uses asynchronous Node JS.

10 seconds has the following intent: a hard state passes validation if its most recent event and the transactuation triggering event are not more than 10 seconds apart.

A sensing policy is an acceptable level of hard-read failures that a transactuation can tolerate. It specifies that under what condition a perform lambda can be executed over a returned list of window-validated sensors. The perform lambda in turn may or may not execute depending on the sensing policy. Transactuations support three sensing policies:

- *All*: ensures that the perform lambda executes only if all hard states in the sensor list pass validation. Consider an application that reads presence sensors of every user and turns on cameras if no one is present. For privacy, *all* sensors need to pass validation. If even one presence sensor fails, it should not risk turning on the cameras since it violates privacy.
- *Any*: guarantees the execution of the perform lambda as long as at least one hard state in the sensor list passes validation. For example, an application that computes average humidity level from multiple sensors to control fans, executes accordingly with correct semantics, even if some sensors fail, but not all.
- *None*: states that the perform lambda executes over the returned validated list of hard states regardless of how many hard states are unavailable.

Observe that a time window along with a sensing policy helps preserve *HR*→* dependency as per the developer's intention to preserve invariant (D1). To preserve invariant (D2),

a developer needs to specify an *actuating policy*. The actuating policy is an acceptable level of hard-write failures that is tolerable. To meet an actuating policy in case of a failure, soft writes inside a transactuation roll back to their initial values, and onFailure lambda executes. Similar to a sensing policy, an actuating policy supports the following semantics:

- *All*: states that modifications to soft states commit if all hard writes successfully finish. An example of this policy is an application that locks all doors and sets home state to `safe`. If even one door fails, the home state should not be set.
- *Any*: guarantees that soft state modifications inside a lambda commits if at least one hard write succeeds. For example, an application that actuates all sirens and sets the flag `ringing`. Even if only one siren rings, the flag should be set.
- *None*: states that soft writes commit despite of failures.

onSuccess lambda. An onSuccess lambda executes if the perform lambda of a transactuation succeeds (i.e., sensing and actuating policies are met). A developer can assign an onSuccess lambda to a transactuation via `onSuccess()` as shown in line 17 of Listing 3.

onFailure lambda. An onFailure lambda executes if a transactuation cannot meet its sensing or actuating policies. It is assigned to a transactuation via `onFailure()` as depicted in line 25 of Listing 3.

When a developer has set up all the lambdas for a transactuation, she executes the transactuation by invoking `execute()` (line 29), which is an asynchronous call that executes the perform lambda in the background.

Listing 3 illustrates the CO_2 Vent rewritten with the transactuation abstraction. The perform lambda is parameterized with 5s time window. The transactuation only reads one hard state, `co2`. The lambda executes if the latest sensor update from `co2`, and the triggering event, which is also `co2` fall in the 5 second time interval. `switches`, which binds to an array of fans, requires the "all" policy if we want the soft writes to be consistent with the actuations. The soft state `active` will be set to true only if all fans can be turned on, otherwise, `active` remains unchanged.

4.2 Chaining transactuations

A transactuation can be chained to other transactuations by invoking it in their onSuccess and onFailure lambdas. As we shall see in the next section, the runtime guarantees to execute chained transactuations sequentially: if a transactuation τ_j is invoked in onSuccess lambda of τ_i, τ_j is guaranteed to see the updates τ_i makes. We call this ordered execution of transactuations as T-Chain. This is particularly relevant in an asynchronous runtime where high latency operations can finish in arbitrary order, executing outside the critical path such as in worker threads [25, 44]. Thus, if τ_j wants to use a soft state written by τ_i, τ_j needs to be invoked in onSuccess lambda of τ_i. In addition, if τ_j requires actuations of τ_i to complete before it, these two transactuations must form a T-Chain.

5 Relacs

In this section, we detail the design of our runtime, called Relacs, that execute smart-home applications, along with a supporting key-value store called *Relacs Store*.

5.1 Relacs Store

All soft and hard states inside a transactuation are stored in a key-value store called Relacs Store. It hides all complexities of working with sensors and actuators by allowing developers to not only perform read/write operations on soft states inside a transactuation, but also to issue hard reads/writes.

Conceptually, every state inside the Relacs Store maintains two values, *speculative* and *final*. A speculative value means that the state has been updated logically in the Relacs Store, but is not confirmed to be final (i.e., issued to an IoT device). For example, a transactuation that wants to unlock a door will have the speculative value of the door set to `unlocked`, before the actuation command succeeds. When Relacs receives an ack event confirming the success of an actuation command, it updates the final value and discards the speculative value. Along with setting the final value, the Relacs Store also logs the timestamp of the ack event for validating a time window of a transactuation reading that hard state. In Section 5.2, we explain how speculative states help Relacs to speculatively execute transactuations.

Since multiple hard writes on the same state can execute before the system receives an ack from the corresponding device, Relacs Store needs to record all versions of speculative values that have not been finalized yet. When reading a state, Relacs Store returns the latest speculative value, or the final value if no speculative value exists. For instance, consider the following transactuations: a transactuation τ_i sets a lamp color to red. While the lamp is changing its color, τ_j changes the lamp color to green. In this example, Relacs Store logs both speculative values. Thus, if τ_k tries to read the state of the lamp, Relacs Store returns green, even if the lamp has not completed executing the first actuation command to change its color to red.

5.2 Execution Model

A transactuation execution model comprises of the following three phases:

1. *Hard read phase*: to start executing a transactuation, the system first needs to determine if it can read the required hard states in the sensor list which satisfy the specified window and the sensing policy. If so, the system proceeds to the next phase. For a poll-based sensor, if Relacs fails to validate the

window, it polls the sensor to check if it can get a fresh value. For a push-based sensor, Relacs simply waits, as long as the window is valid, to receive an event from the sensor. Observe that the window is valid as long as the specified time window has not passed since the transactuation triggering event. If the window becomes invalid, and the list of received events fails staleness validation, it cannot execute the perform lambda, and proceeds to execute the onFailure lambda.

2. *Speculative Commit Phase*: since IoT devices cannot roll back, Relacs needs to make sure that a transactuation will definitely commit before performing real actuations. Therefore, it employs a speculative execution model where a perform lambda first executes speculatively, without performing any real actuation. Once the perform lambda finishes, it tries to speculatively commit like a normal transaction inside Relacs Store. Therefore, new speculative values are committed for modified soft and hard states. Additionally, committing new speculative values may trigger other handler functions subscribed to these states. Finally, Relacs starts executing the onSuccess lambda of the transactuation when it commits. Note that these lambdas triggered by speculative commit execute their transactuations speculatively.

3. *Final Commit Phase*: in the last phase, Relacs sends actuation commands that correspond to hard writes. A transactuation τ_i can start its final phase, when the following three conditions hold: first, all transactuations that precede τ_i in the T-Chain finally commit. Second, all transactuations updating states that τ_i read, finally commit. Third, no other finally committing τ_j conflicts with τ_i. More specifically, the readset of τ_i does not have any intersection with the writeset of some finally committing transactuation, and the writeset of τ_i does not intersect with both readset and writeset of some finally committing transactuation.

Relacs finally commits the transactuation when sufficient acks are received from actuators to satisfy its actuating policy. If the transactuation times out without satisfying its actuating policy, all soft writes inside the transactuation roll back to their initial state, and the transactuation finally commits. Next, onFailure lambda executes if it has been defined. Moreover, all speculative transactuations invoked by the failed transactuation abort (e.g., chained transactuations), and transactuations that bear data dependencies with the failed transactuation need to re-execute.

5.3 Relacs Runtime

Relacs is built atop serverless computing [32, 42]. The runtime comprises two classes of functions namely application functions and system functions. We explain these functions in detail here.

Application Functions. An application can comprise several handlers which are triggered when particular states in the Relacs Store change (publish-subscribe model), and each handler can comprise several transactuations. An application submitted to run by Relacs system is transformed into a set of application functions to run on serverless instances as follows:

1. For each handler, Relacs transforms the logic of an embedded transactuation (i.e., perform lambda) into a transaction that can execute transactionally inside the Relacs Store.
2. The logic inside onSuccess lambda and onFailure lambda are transformed into stand-alone serverless functions called *success* and *failure* functions, respectively, hereafter. If onSuccess lambda or onFailure lambda is comprised of transactuations with their own onSuccess lambda and onFailure lambda (T-Chain), the transformations are applied recursively.
3. Finally, every handler is transformed into a runnable stand-alone serverless function, called *handler* function.

System Functions. Relacs comprises a serverless function called *updater* function that is invoked whenever the state of a sensor or an actuator changes. Upon receiving a notification, the updater updates the hard state corresponding to the event in Relacs Store, and launches an instance of subscribed handler function(s).

Final-committer is a designated function to perform the final commits. It selects speculative transactuations that can finally commit without breaking the final commit rules, issues all of their actuation commands, and marks the actuations as issued. When a successful actuation receives a notification (ack) from an IoT device, the updater function updates its corresponding state in Relacs Store, and marks the actuation command as done transactionally.

In order to detect an actuation failure, Relacs has a *failure-detector* function that runs periodically, and checks whether an ack is received for an actuation command. If after certain threshold no ack is received, the failure detector marks the actuation as failed. If actuating policy is not met, the enclosing transactuation commits with rollback of soft writes, which triggers a *re-executor* function to re-execute transactuations that have data dependencies with the failed transactuation.

5.4 Fault Tolerance

A function in serverless computing is not guaranteed to complete, and can terminate at any arbitrary point of execution. Yet, Relacs guarantees applications to execute reliably despite failures as follows.

Relacs ensures that all transactuations are executed exactly-once even if an application function (handler, success, or failure) fails during its execution. To this end, Relacs maintains two logs: function log and transactuation log. Function log is a write-ahead log for application functions. The function name along with ID of the triggering event is recorded in the function log before the function executes. Transactuation log atomically records a transactuation name and the event ID during the speculative commit of a transactuation along with updates to soft/hard states.

A system function called *serverless checker* runs periodically, and inspects the function log to execute functions which have failed. In either case, the serverless checker invokes the failed functions again. This might lead to duplicated executions of transactuations that have executed. To prevent this, Relacs checks if a particular transactuation is in the transactuation log, and skips its execution if present. [1]

Currently, the updater failure is treated as an equivalent of sensor or actuator failure and it is handled by transactuation semantics. To address final committer failure, Relacs runs the final committer periodically to complete pending final commits by actuating unissued actuations. To preclude contention between the periodic and the regular final committer that can run concurrently, Relacs uses leases and ETAGS à la Tuba [21] in the final committer to ensure correctness.

5.5 Implementation

We implemented Relacs runtime and Relacs Store on top of Microsoft Azure. We used Azure Function (serverless computing) to implement the runtime, and used Azure Cosmos DB to build Relacs Store. All serverless functions were implemented with Azure Function. Application functions are triggered by HTTP calls and system functions are triggered on Cosmos DB updates or periodic timers. The parts of the protocol that need to update Relacs Store transactionally (including perform lambda) are transformed into Cosmos DB stored procedures [3].

Currently, Relacs has only been integrated with Samsung SmartThings. SmartThings allows a developer to build a web service that connects with devices in a home [18]. We built a gateway that forwards actuation commands from Relacs to actuators and also polls sensor data.

5.6 Discussion

As described, Relacs validates sensor failures through event timestamps and actuator failures through timeouts. For sensor validation, as explained, if validation fails and a device is pollable, Relacs polls the device within the window constraints. If a device is push-based but pollable, Relacs polls the device and if the validation fails again, it waits for its push-interval within the time window. However, if the device is purely push-based, Relacs cannot differentiate between inactivity and failure. We inspected 188 SmartThings-compatible devices and found that 113 of them are pollable. Likewise, actuation failures are detected with timeouts, first on initial ack from smart-home connector, followed by notification on final actuator state change. Again, if the ack message is lost, Relacs can incorrectly rollback soft states. However, transactuations can still help developers to prioritize home safety over convenience such as always setting a soft state to a conservative value; e.g., in Smart Security (Listing 2) to ensure that the alarm eventually rings.

[1] Note that any failure during the speculative commit results in a regular transactional abort and transactuation log is not updated. Hence the transactuation is retried when the function reexecutes.

6 Evaluation

In this section, we report our evaluation results on programmability, effectiveness of transactuations in enforcing correctness, and the overhead incurred by Relacs to provide transactuation semantics.

We selected 10 SmartThings applications from the applications that we statically analyzed. These applications are publicly available on SmartThings repository [19]. The applications cover the four most common categories—Security (Sc), Safety (Sf), Convenience (Cn), and Energy Efficiency (Ee). Instead of using the original version that runs on SmartThings cloud, we implemented the following three versions of the applications, that run on Azure Functions, using Javascript Node JS [44]. This allows us to compare an application with transactuations against an application without transactuations in an apple-to-apple fashion.

- *BE:* we wrote a best-effort version (BE) of the applications without the transactuation abstraction. The BE version follows the default semantics that ignores device failure, exactly-once execution, and isolation.
- *BE+Con:* since the BE version ignores potential failures in devices or applications, we implemented a *best-effort with consistency* (BE+Con) version of an application which adds code that keeps device states consistent with application states. More specifically, BE+Con introduces both sensor window validation and soft state rollback code. However, it ignores the isolation guarantee that transactuations provide.
- *TN:* we also implemented these applications with the transactuation abstraction (TN). 5 applications out of the evaluated 10 applications used T-Chain to establish order among hard and soft states.

Experimental setup. We set up SmartThings compatible devices and measured the round trip latency of four devices in a typical smart home: a door lock, a bulb, a power strip, and a smart power plug. The door lock has a significant latency of nearly 3.6s on average and maximum of nearly 9.8s, over 100 trials. The other devices incur an average latency of nearly 0.7s with the maximum at nearly 3.7s. Since we had a limited set of devices, we parallelized our experiments by simulating the devices using latency data on a Raspberry Pi Model 3 [13]. It comes with a 1.2 GHz 32-bit quadcore ARM Cortex-A53 processor and 1 GB RAM. In addition, the simulator also allowed us to easily inject failures for our experiments.

Application	#HR	#HW	Transactuation Policy	LOC		
				BE	BE+Con	TN
Rise And Shine (Cn1)	1 (*)	1	2 (none, none)	72	195	68
Whole House Fan (Cn2)	1 (*), 3	2 (*)	1 (none, none)	29	176	26
Thermostat Auto Off (Cn3)	1 (*)	2	1 (all, none), 1 (all, all), 1 (none, all)	70	198	68
Auto Humidity Vent (Ee1)	1 (*), 1	3(*), 1	1 (any, none), 1 (none, any), 1 (none, none), 1 (all, any)	49	170	100
Lights Off With No Motion (Ee2)	1 (*), 1	1 (*)	2 (all, all)	56	161	67
Cameras On When Away (Sc1)	2 (*)	2 (*)	1 (all, none), 1 (any, none)	31	149	88
Nobody Home (Sc2)	1 (*)	1	1 (all, none), 1 (any, none), 1 (none, none)	65	175	62
Smart Security (Sc3)	2 (*)	2 (*)	1 (all, all)	144	323	144
CO2 Vent (Sf1)	1	2 (*)	1 (all, all)	29	152	26
Lock It When I Leave (Sf2)	3 (*)	2 (*), 2	2 (none, none), 1 (all, none)	51	180	54

Table 2: Properties of each benchmark application including the number of hard reads and hard writes (* denotes an operation to an array of devices with a single command, for example, 2 (*) means 2 operations, each accessing a device group); the fault-tolerance policies for the TN configuration in a format of (sensing, actuating) (Col 4); and programability shown by LOC comparison among transactuation (TN), best effort (BE), and best effort with consistency (BE+Con) (Col 5).

6.1 Programmability

In order to evaluate the programmability and convenience of using transactuation in contrast to manually writing failure handling code, we compare lines of code (LOC) of applications, using CLOC [6].

Table 2 shows the programmability evaluation (LOC) along with the number of hard reads and writes, and transactuation policies we employ for each application. Observe that TN and BE versions are comparable in LOC despite no guarantees in the BE version, except in Ee1 where we introduce new soft states and four transactuations, each part of T-Chains, in order to ensure consistency. BE+Con version requires substantial code to explicitly handle failures. As mentioned earlier, BE+Con version validates sensor freshness similar to transactuation and may roll back soft states after determining the outcome of actuations for hard write to soft write dependencies. Finally, although transactuations require more code in order to create T-Chains, it automatically handles failure, and simplifies writing reliable applications considerably.

6.2 Correctness

Table 3 shows the applications that we evaluated with their inherent undesirable behaviors on transient or longer duration failures. The second column shows the undesirable behaviors, and the third column shows the outcome of using transactuations. The last column explains the mechanism transactuations use to resolve or mitigate the issue. We considered different types of failures that transactuations can address (i.e., unavailable sensors and failed actuations), and injected these failures by dropping event or actuation messages. Transactuation addresses these issues with three techniques. First, sensor staleness validation prevents the execution of perform lambda and executes onFailure lambda that can notify a user. Second, actuation losses are detected automatically and associated soft writes are rolled back to ensure consistency. Third, when one actuation depends on another, we used an intermediate soft state to chain two transactuations each having actuations. For example, in Sc3 (Smart Security) application, inconsistency between the alarm actuation and the soft write is resolved using roll back to eliminate the issue. However, some applications need to use multiple chained transactuations to correctly address actuation dependencies.

6.3 Overhead

To evaluate the overhead of transactuations, we measured execution time of the applications as follows. We started timing when an application began executing, and stopped when every soft write committed and all actuations completed. Our performance results are summarized in Figure 1. Each value is the mean of 30 runs, with 95% confidence intervals.

Failure-free. We first compare the execution times of TN and BE versions without any injected failures. The overhead of transactuations is attributed to (1) safeguarding against inconsistencies due to inherently concurrent execution, (2) providing fault tolerance, and (3) enforcing actuation orders of T-Chains. We note that the final committer function imposes significant overhead on Relacs since it is invoked[2] automatically by CosmosDB updates. For instance, we observed that its start may be delayed between zero to five seconds. The periodic final committer which we set to run every second helps to mitigate this overhead.

Figure 1a shows that, on average (geomean), the TN version incurs 1.5 times slowdown compared to BE. Observe that the

[2]Other functions except the re-executor are invoked by HTTP calls.

App	Undesirable consequence	Transactuation effect	Mechanism used
Cn1	Mode not set permanently	✓	Soft state rollback
Cn2	Incorrect behavior Fans not ON irreversibly	Issue detected and user notified ✓	Sensor staleness validation Soft state rollback
Cn3	Thermostat not OFF Incorrect mode	✓ ✓	Soft state rollback Soft state rollback
Ee1	Incorrect energy and operation time reported Incorrect behavior	✓ Issue detected and user notified	Soft state rollback and chaining Sensor staleness validation
Ee2	Incorrectly turning lights ON/OFF	Issue detected and user notified	Sensor staleness validation
Sc1	Incorrect behavior Actuation failure	Issue detected and user notified ✓	Sensor staleness validation Chaining
Sc2	Incorrect mode set Home mode change w/o notification	Issue detected and user notified ✓	Sensor staleness validation soft state rollback
Sc3	Intruder motion not detected Alarm not active irreversibly	Issue detected and user notified ✓	Sensor staleness validation soft state rollback
Sf1	Incorrect behavior Exhauts not ON irreversibly	Issue detected and user notified ✓	Sensor staleness validation soft state rollback
Sf2	Door unlocked but home vacant Door locked at arrival	Issue detected and user notified ✓	Sensor staleness validation Chaining

Table 3: Applications with undesirable consequences on induced failures. Column 3 shows failure avoidance or mitigation when written with transactuations. Column 4 shows the internal mechanism used by the transactuations. A checkmark implies that transactuation automatically resolves the issue.

speculative commit duration (TN.SC) is significantly smaller than the final commit duration (TN.FC). Figure 1a also breaks down the final commit time into actuation time (TN.FC.ACT) and the final-committer triggering overhead (TN.FC.TRIG). As mentioned earlier, the triggering overhead is significantly large, especially, in the case of a long T-Chain like Ee1 (4 transactuations).

With failure. In this scenario, we conducted two experiments. In each experiment, we used a dummy application that issued a dummy actuation, and updated a dummy soft state. In the first experiment, the dummy actuation turned on a smart switch (low-latency actuation). In the second one, it actuated a door lock (high-latency actuation). We introduced an artificial data dependency (RAW) by forcing all benchmark applications to read the dummy soft state before executing their core logic. Lastly, we injected a failure to the dummy actuation to trigger failure detection and handling in the dummy application and re-execution of the benchmark applications to repair the broken data dependency. Because devices have different actuation latencies, the timeout thresholds to declare failed actuations are specific to each device. More specifically, we used the maximum observed latency for each device (i.e., 4s for the smart switch and 10s for the door lock).

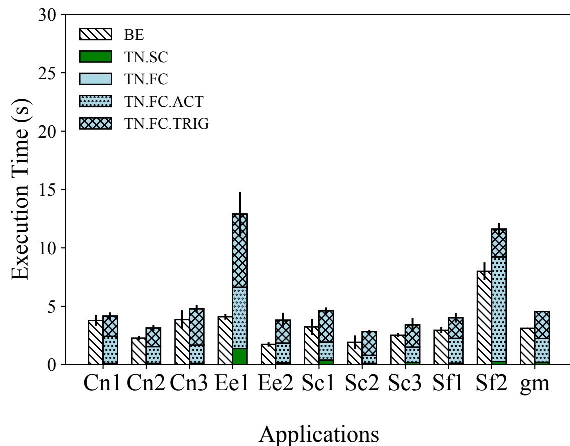
Figure 1b compares the execution time of the failure-free case against the two failure experiments. The additional overhead we observe here is the failure detection overhead which includes the timeout (TN.FD.TO) and the overhead of triggering the re-executor function (TN.FD.TRIG). Similar to the final committer, the re-executor is invoked automatically by Cosmos DB when actuations are marked as `failed`, thus it incurs similar overhead. Observe that the failure experiments have two stacked bars of speculative commits. The second bar shows the re-execution of transactuations with broken dependencies.

As expected, introducing a failure results in longer execution times for the applications. This slowdown is caused by the timeout threshold plus the re-executor triggering overhead (~2s). Moreover, the difference between the middle and right bars for each application is the difference in timeout thresholds for low and high latency actuations (~6s).

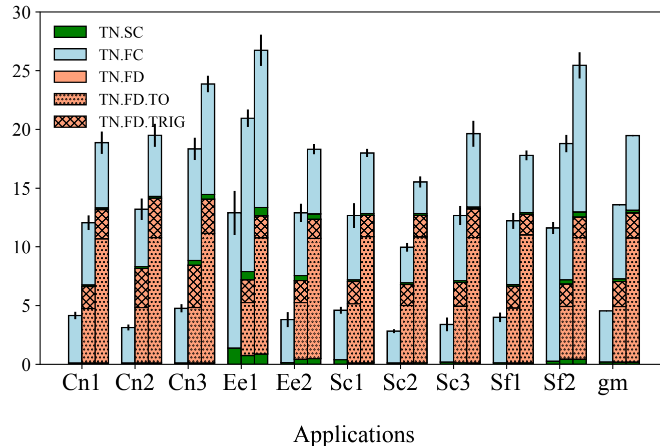
7 Related Work

Checking Correctness. Soteria [22] employs model checking to identify contradicting interactions between IoT applications. For example, water leak detection turns off a water valve while smoke detection attempts to turn on a fire sprinkler. Prior work like DeLorean [24] models absolute and relative time to find timing bugs in event driven programs, e.g., door open at unsafe times. In contrast, our work tackles a different problem, the lack of reliability and isolation, using a dynamic technique. IoT analyses also use dynamic taint analyses like techniques to detect source of security breaches [46] and dynamic program slicing to explain behaviors [40]. We use static dependence analysis to report potential problems.

Programming abstractions. Using speculative execution for improving latency and performance is a common technique in many transactional and replicated systems. These can be classified into two categories: systems [34, 41, 47]



(a) Execution times for BE and TN versions in failure-free case. We break down the execution time of TN into speculative commit (TN.SC) and final commit (TN.FC). TN.FC is shown as actuation time (TN.FC.ACT) and as overhead to trigger the final-committer function (TN.FC.TRIG).



(b) Execution time comparison for failure-free and failure cases. For each application, we show 3 bars, failure-free case (the left bar), low-latency actuation failure case (the middle bar), and high-latency actuation failure case (the right bar). For the failure cases, the breakdown includes failure detection time (TN.FD) which is subdivided into timeout detection (TN.FD.TO) and re-execution triggering overhead (TN.FD.TRIG).

Figure 1: The execution time of 10 applications chosen from SmartThings repository and their geomean (gm) for BE and TN versions of applications in failure-free and failure scenarios.

that hide the effects of speculation from applications, and work [29, 31, 43] that expose speculation results to applications. While certain applications in the latter case can benefit by reading speculative values, they need to handle possible side effects of acting on misspeculated values. With Relacs, effects of speculatively committed transactuations are exposed to other transactuations. Yet, no transactuation can finally commit, and actuate devices until all transactuations that it speculatively read from finally commit.

Planet [43] provides a mechanism to speculate on partial state of a transaction in distributed environments. The abstraction allows a developer to continue based on a predictive outcome, and later receive a confirmation or an apology. In contrast, we target a different environment and problem, and provide a simplified way to address device failure handling.

Execution semantics and conflict detection. IOTA [40] defines a calculus for programs in IoT domain. They also define an execution semantics to eliminate races on actions against the same physical event. Similar races can be resolved in our system by reordering transactuations according to programmer annotations similar to Zave et al. [48]. IOTA also shows offline analyses to detect device conflicts. Conflict detection in a home can include static model checking [38] or dynamic analyses [48] to detect feature interactions [38] and accesses to the same device [26]. They detect commands due to single event or concurrent independent events to the same device, e.g., simultaneous turning on and off on a device. The execution semantics of our system provides isolation naturally and can easily be enhanced to report device interactions by intersecting read-write sets of transactuations dynamically.

8 Conclusion

In this paper, we identified a fundamental problem that arises due to failures in IoT systems that interact with the physical world. We analyzed smart-home applications, and showed how application semantics is broken due to different failures that occur in an IoT environment. We introduced an abstraction, called transactuation, that allows a developer to build reliable IoT applications. Our runtime, called Relacs, enforces the semantic guarantees of transactuations. Our evaluation demonstrated programmability, performance, and effectiveness of the transactuation abstraction on top of our runtime.

9 Acknowledgment

We would like to thank our shepherd, Gernot Heiser and anonymous reviewers for their insightful and valuable feedback. We would also like to thank Nitin Agrawal, Arani Bhattacharya, Juan Colmenares, Iqbal Mohomed, Marc Shapiro, Pierre Sutra, Ahmad Bisher Tarakji, and Ashish Vulimiri for their suggestions and helpful discussions.

References

[1] Arrow functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions.

[2] AWS Lambda Retry Behavior. https://docs.aws.amazon.com/lambda/latest/dg/retries-on-errors.html.

[3] Azure Cosmos DB server-side programming: Stored procedures, database triggers, and UDFs. https://docs.microsoft.com/en-us/azure/cosmos-db/programming.

[4] Bluetooth Low Energy. https://www.bluetooth.com.

[5] CO2 Vent. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps/dianoga/co2-vent.src.

[6] Count Lines of Code. http://cloc.sourceforge.net.

[7] Expressions. https://docs.python.org/2/reference/expressions.html.

[8] Groovy ast interface. http://docs.groovy-lang.org/docs/groovy-2.4.0/html/api/org/codehaus/groovy/ast/package-summary.html.

[9] Inconsistent Behavior. https://community.smartthings.com/t/inconsistent-behavior/35284.

[10] IoTBench-test-suite. https://github.com/IoTBench/IoTBench-test-suite/tree/master/openHAB.

[11] Lambda Expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[12] OpenHAB: Empowering the Smart Home. https://www.openhab.org.

[13] Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[14] SmartThings. http://www.smartthings.com/.

[15] SSA1 / SSA2 Instruction Manual. https://support.smartthings.com/hc/en-us/article_attachments/200715310/ssa_manual_14may2011_-_new_address0.pdf.

[16] Z-Wave Alliance. http://www.z-wavealliance.org.

[17] ZigBee Alliance. http://www.zigbee.org/.

[18] Web Services SmartThings. https://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/index.html, 2018.

[19] SmartThings Smart Apps. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps, 2019.

[20] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. Rivulet: A Fault-tolerant Platform for Smart-home Applications. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference (MIDDLEWARE '17)*, Las Vegas, NV, December 2017.

[21] Masoud Saeida Ardekani and Douglas B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[22] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT Safety and Security Analysis. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, July 2018.

[23] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. *Rice University, CS Technical Report 06-33870*, January 2001.

[24] Jason Croft, Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. Systematically Exploring the Behavior of Control Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*, Santa Clara, CA, July 2015.

[25] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the 2017 European Conference on Computer Systems (EuroSys '17)*, Belgrade, Serbia, April 2017.

[26] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An Operating System for the Home. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[27] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[28] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, September 2013.

[29] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[30] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC '14)*, Seattle, WA, November 2014.

[31] Pat Helland and Dave Cambell. Building on Quicksand. In *Proceedings of the 4th Conference on Innovative Data Systems Research (CIDR '09)*, Pacific Grove, CA, January 2009.

[32] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, Denver, CO, June 2016.

[33] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys '11)*, Seattle, WA, November 2011.

[34] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012.

[35] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*, Austin, TX, May 2016.

[36] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 2013 European Conference on Computer Systems (EuroSys '13)*, Prague, Czech Republic, April 2013.

[37] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[38] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: Building an Internet of Safe Things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN '15)*, Seattle, WA, April 2015.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS '08)*, Seattle, WA, March 2008.

[40] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD '17)*, Vancouver, Canada, October 2017.

[41] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 191–205, 2005.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, MA, July 2018.

[43] Gene Pang, Tim Kraska, Michael J. Franklin, and Alan Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, Snowbird, UT, June 2014.

[44] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, November 2010.

[45] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical Trigger-Action Programming in the Smart Home. In *Proceedings of the 2014 SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, Toronto, Canada, April 2014.

[46] Qi Wang, Wajih Ul Hassan, Adam M. Bates, and Carl A. Gunter. Fear and Logging in the Internet of Things. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, (NDSS '18)*, San Diego, CA, Februay 2018.

[47] Benjamin Wester, James A. Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, April 2009.

[48] Pamela Zave, Eric Cheung, and Svetlana Yarosh. Toward user-centric feature composition for the Internet of Things. *arXiv preprint arXiv:1510.06714*, October 2015.

Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha
University of Massachusetts Amherst

Abstract

All major web browsers now support WebAssembly, a low-level bytecode intended to serve as a compilation target for code written in languages like C and C++. A key goal of WebAssembly is performance parity with native code; previous work reports near parity, with many applications compiled to WebAssembly running on average 10% slower than native code. However, this evaluation was limited to a suite of scientific kernels, each consisting of roughly 100 lines of code. Running more substantial applications was not possible because compiling code to WebAssembly is only part of the puzzle: standard Unix APIs are not available in the web browser environment. To address this challenge, we build BROWSIX-WASM, a significant extension to BROWSIX [29] that, for the first time, makes it possible to run unmodified WebAssembly-compiled Unix applications directly inside the browser. We then use BROWSIX-WASM to conduct the first large-scale evaluation of the performance of WebAssembly vs. native. Across the SPEC CPU suite of benchmarks, we find a substantial performance gap: applications compiled to WebAssembly run slower by an average of 45% (Firefox) to 55% (Chrome), with peak slowdowns of 2.08× (Firefox) and 2.5× (Chrome). We identify the causes of this performance degradation, some of which are due to missing optimizations and code generation issues, while others are inherent to the WebAssembly platform.

1 Introduction

Web browsers have become the most popular platform for running user-facing applications, and until recently, JavaScript was the only programming language supported by all major web browsers. Beyond its many quirks and pitfalls from the perspective of programming language design, JavaScript is also notoriously difficult to compile efficiently [12, 17, 30, 31]. Applications written in or compiled to JavaScript typically run much slower than their native counterparts. To address this situation, a group of browser vendors jointly developed *WebAssembly*.

WebAssembly is a low-level, statically typed language that does not require garbage collection, and supports interoperability with JavaScript. The goal of WebAssembly is to serve as a universal compiler target that can run in a browser [15, 16, 18].[1] Towards this end, WebAssembly is designed to be fast to compile and run, to be portable across browsers and architectures, and to provide formal guarantees of type and memory safety. Prior attempts at running code at native speed in the browser [4, 13, 14, 38], which we discuss in related work, do not satisfy all of these criteria.

WebAssembly is now supported by all major browsers [8, 34] and has been swiftly adopted by several programming languages. There are now backends for C, C++, C#, Go, and Rust [1, 2, 24, 39] that target WebAssembly. A curated list currently includes more than a dozen others [10]. Today, code written in these languages can be safely executed in browser sandboxes across any modern device once compiled to WebAssembly.

A major goal of WebAssembly is to be faster than JavaScript. For example, the paper that introduced WebAssembly [18] showed that when a C program is compiled to WebAssembly instead of JavaScript (`asm.js`), it runs 34% faster in Google Chrome. That paper also showed that the performance of WebAssembly is competitive with native code: of the 24 benchmarks evaluated, the running time of seven benchmarks using WebAssembly is within 10% of native code, and almost all of them are less than 2× slower than native code. Figure 1 shows that WebAssembly implementations have continuously improved with respect to these benchmarks. In 2017, only seven benchmarks performed within 1.1× of native, but by 2019, this number increased to 13.

These results appear promising, but they beg the question: *are these 24 benchmarks representative of WebAssembly's intended use cases*?

[1] The WebAssembly standard is undergoing active development, with ongoing efforts to extend WebAssembly with features ranging from SIMD primitives and threading to tail calls and garbage collection. This paper focuses on the initial and stable version of WebAssembly [18], which is supported by all major browsers.

The Challenge of Benchmarking WebAssembly The aforementioned suite of 24 benchmarks is the PolybenchC benchmark suite [5], which is designed to measure the effect of polyhedral loop optimizations in compilers. All the benchmarks in the suite are small scientific computing kernels rather than full applications (e.g., matrix multiplication and LU Decomposition); each is roughly 100 LOC. While WebAssembly is designed to accelerate scientific kernels on the Web, it is also explicitly designed for a much richer set of full applications.

The WebAssembly documentation highlights several intended use cases [7], including scientific kernels, image editing, video editing, image recognition, scientific visualization, simulations, programming language interpreters, virtual machines, and POSIX applications. Therefore, WebAssembly's strong performance on the scientific kernels in PolybenchC do not imply that it will perform well given a different kind of application.

We argue that a more comprehensive evaluation of WebAssembly should rely on an established benchmark suite of large programs, such as the SPEC CPU benchmark suites. In fact, the SPEC CPU 2006 and 2017 suite of benchmarks include several applications that fall under the intended use cases of WebAssembly: eight benchmarks are scientific applications (e.g., `433.milc`, `444.namd`, `447.dealII`, `450.soplex`, and `470.lbm`), two benchmarks involve image and video processing (`464.h264ref` and `453.povray`), and all of the benchmarks are POSIX applications.

Unfortunately, it is not possible to simply compile a sophisticated native program to WebAssembly. Native programs, including the programs in the SPEC CPU suites, require operating system services, such as a filesystem, synchronous I/O, and processes, which WebAssembly and the browser do not provide. The SPEC benchmarking harness itself requires a file system, a shell, the ability to spawn processes, and other Unix facilities. To overcome these limitations when porting native applications to the web, many programmers painstakingly modify their programs to avoid or mimic missing operating system services. Modifying well-known benchmarks, such as SPEC CPU, would not only be time consuming but would also pose a serious threat to validity.

The standard approach to running these applications today is to use Emscripten, a toolchain for compiling C and C++ to WebAssembly [39]. Unfortunately, Emscripten only supports the most trivial system calls and does not scale up to large-scale applications. For example, to enable applications to use synchronous I/O, the default Emscripten `MEMFS` filesystem loads the entire filesystem image into memory before the program begins executing. For SPEC, these files are too large to fit into memory.

A promising alternative is to use BROWSIX, a framework that enables running unmodified, full-featured Unix applications in the browser [28, 29]. BROWSIX implements a Unix-compatible kernel in JavaScript, with full support for

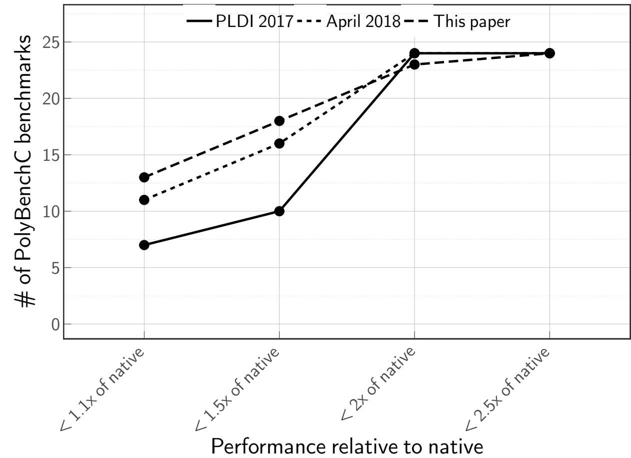


Figure 1: Number of PolyBenchC benchmarks performing within $x\times$ of native. In 2017 [18], seven benchmarks performed within $1.1\times$ of native. In April 2018, we found that 11 performed within $1.1\times$ of native. In May 2019, 13 performed with $1.1\times$ of native.

processes, files, pipes, blocking I/O, and other Unix features. Moreover, it includes a C/C++ compiler (based on Emscripten) that allows programs to run in the browser unmodified. The BROWSIX case studies include complex applications, such as LaTeX, which runs entirely in the browser without any source code modifications.

Unfortunately, BROWSIX is a JavaScript-only solution, since it was built before the release of WebAssembly. Moreover, BROWSIX suffers from high performance overhead, which would be a significant confounder while benchmarking. Using BROWSIX, it would be difficult to tease apart the poorly performing benchmarks from performance degradation introduced by BROWSIX.

Contributions

- **BROWSIX-WASM:** We develop BROWSIX-WASM, a significant extension to and enhancement of BROWSIX that allows us to compile Unix programs to WebAssembly and run them in the browser with no modifications. In addition to integrating functional extensions, BROWSIX-WASM incorporates performance optimizations that drastically improve its performance, ensuring that CPU-intensive applications operate with virtually no overhead imposed by BROWSIX-WASM (§2).

- **BROWSIX-SPEC:** We develop BROWSIX-SPEC, a harness that extends BROWSIX-WASM to allow automated collection of detailed timing and hardware on-chip performance counter information in order to perform detailed measurements of application performance (§3).

- **Performance Analysis of WebAssembly:** Using BROWSIX-WASM and BROWSIX-SPEC, we conduct the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that WebAssembly does run faster than JavaScript (on average 1.3× faster across SPEC CPU). However, contrary to prior work, we find a substantial gap between WebAssembly and native performance: code compiled to WebAssembly runs on average 1.55× slower in Chrome and 1.45× slower in Firefox than native code (§4).

- **Root Cause Analysis and Advice for Implementers:** We conduct a forensic analysis with the aid of performance counter results to identify the root causes of this performance gap. We find the following results:

  1. The instructions produced by WebAssembly have more loads and stores than native code (2.02× more loads and 2.30× more stores in Chrome; 1.92× more loads and 2.16× more stores in Firefox). We attribute this to reduced availability of registers, a sub-optimal register allocator, and a failure to effectively exploit a wider range of x86 addressing modes.
  2. The instructions produced by WebAssembly have more branches, because WebAssembly requires several dynamic safety checks.
  3. Since WebAssembly generates more instructions, it leads to more L1 instruction cache misses.

  We provide guidance to help WebAssembly implementers focus their optimization efforts in order to close the performance gap between WebAssembly and native code (§5, 6).

BROWSIX-WASM and BROWSIX-SPEC are available at `https://browsix.org`.

2 From BROWSIX to BROWSIX-WASM

BROWSIX [29] mimics a Unix kernel within the browser and includes a compiler (based on Emscripten [33, 39]) that compiles native programs to JavaScript. Together, they allow native programs (in C, C++, and Go) to run in the browser and freely use operating system services, such as pipes, processes, and a filesystem. However, BROWSIX has two major limitations that we must overcome. First, BROWSIX compiles native code to JavaScript and not WebAssembly. Second, the BROWSIX kernel has significant performance issues. In particular, several common system calls have very high overhead in BROWSIX, which makes it hard to compare the performance of a program running in BROWSIX to that of a program running natively. We address these limitations by building a new in-browser kernel called BROWSIX-WASM, which supports WebAssembly programs and eliminates the performance bottlenecks of BROWSIX.

Emscripten Runtime Modifications BROWSIX modifies the Emscripten compiler to allow processes (which run in WebWorkers) to communicate with the BROWSIX kernel (which runs on the main thread of a page). Since BROWSIX compiles native programs to JavaScript, this is relatively straightforward: each process' memory is a buffer that is shared with the kernel (a `SharedArrayBuffer`), thus system calls can directly read and write process memory. However, this approach has two significant drawbacks. First, it precludes growing the heap on-demand; the shared memory must be sized large enough to meet the high-water-mark heap size of the application for the entire life of the process. Second, JavaScript contexts (like the main context and each web worker context) have a fixed limit on their heap sizes, which is currently approximately 2.2 GB in Google Chrome [6]. This cap imposes a serious limitation on running multiple processes: if each process reserves a 500 MB heap, BROWSIX would only be able to run at most four concurrent processes. A deeper problem is that WebAssembly memory cannot be shared across WebWorkers and does not support the `Atomic` API, which BROWSIX processes use to wait for system calls.

BROWSIX-WASM uses a different approach to process-kernel communication that is also faster than the BROWSIX approach. BROWSIX-WASM modifies the Emscripten runtime system to create an auxiliary buffer (of 64MB) for each process that is shared with the kernel, but is distinct from process memory. Since this auxiliary buffer is a `SharedArrayBuffer` the BROWSIX-WASM process and kernel can use `Atomic` API for communication. When a system call references strings or buffers in the process's heap (e.g., `writev` or `stat`), its runtime system copies data from the process memory to the shared buffer and sends a message to the kernel with locations of the copied data in auxiliary memory. Similarly, when a system call writes data to the auxiliary buffer (e.g., `read`), its runtime system copies the data from the shared buffer to the process memory at the memory specified. Moreover, if a system call specifies a buffer in process memory for the kernel to write to (e.g., `read`), the runtime allocates a corresponding buffer in auxiliary memory and passes it to the kernel. In case the system call is either reading or writing data of size more than 64MB, BROWSIX-WASM divides this call into several calls such that each call only reads or writes at maximum 64MB of data. The cost of these memory copy operations is dwarfed by the overall cost of the system call invocation, which involves sending a message between process and kernel JavaScript contexts. We show in §4.2.1 that BROWSIX-WASM has negligible overhead.

Performance Optimization While building BROWSIX-WASM and doing our preliminary performance evaluation,

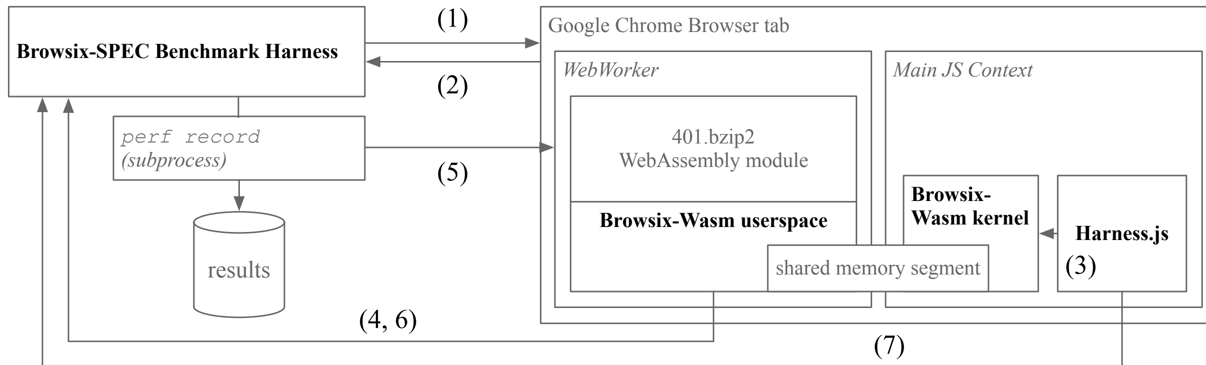


Figure 2: The framework for running SPEC benchmarks in browsers. **Bold** components are new or heavily modified (§3).

we discovered several performance issues in parts of the BROWSIX kernel. Left unresolved, these performance issues would be a threat to the validity of a performance comparison between WebAssembly and native code. The most serious case was in the shared filesystem component included with BROWSIX/BROWSIX-WASM, BROWSERFS. Originally, on each append operation on a file, BROWSERFS would allocate a new, larger buffer, copying the previous and new contents into the new buffer. Small appends could impose substantial performance degradation. Now, whenever a buffer backing a file requires additional space, BROWSERFS grows the buffer by at least 4 KB. This change alone decreased the time the `464.h264ref` benchmark spent in BROWSIX from 25 seconds to under 1.5 seconds. We made a series of improvements that reduce overhead throughout BROWSIX-WASM. Similar, if less dramatic, improvements were made to reduce the number of allocations and the amount of copying in the kernel implementation of pipes.

3 BROWSIX-SPEC

To reliably execute WebAssembly benchmarks while capturing performance counter data, we developed BROWSIX-SPEC. BROWSIX-SPEC works with BROWSIX-WASM to manage spawning browser instances, serving benchmark assets (e.g., the compiled WebAssembly programs and test inputs), spawning `perf` processes to record performance counter data, and validating benchmark outputs.

We use BROWSIX-SPEC to run three benchmark suites to evaluate WebAssembly's performance: SPEC CPU2006, SPEC CPU2017, and PolyBenchC. These benchmarks are compiled to native code using Clang 4.0, and WebAssembly using BROWSIX-WASM. We made no modifications to Chrome or Firefox, and the browsers are run with their standard sandboxing and isolation features enabled. BROWSIX-WASM is built on top of standard web platform features and requires no direct access to host resources – instead, benchmarks make standard HTTP requests to BROWSIX-SPEC.

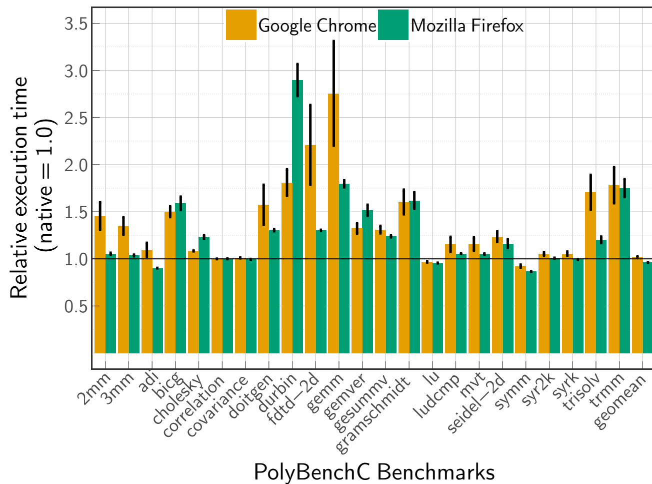
3.1 BROWSIX-SPEC Benchmark Execution

Figure 2 illustrates the key pieces of BROWSIX-SPEC in play when running a benchmark, such as `401.bzip2` in Chrome. First (1), the BROWSIX-SPEC benchmark harness launches a new browser instance using a WebBrowser automation tool, Selenium.[2] (2) The browser loads the page's HTML, harness JS, and BROWSIX-WASM kernel JS over HTTP from the benchmark harness. (3) The harness JS initializes the BROWSIX-WASM kernel and starts a new BROWSIX-WASM process executing the `runspec` shell script (not shown in Figure 2). `runspec` in turn spawns the standard `specinvoke` (not shown), compiled from the C sources provided in SPEC 2006. `specinvoke` reads the `speccmds.cmd` file from the BROWSIX-WASM filesystem and starts `401.bzip2` with the appropriate arguments. (4) After the WebAssembly module has been instantiated but before the benchmark's `main` function is invoked, the BROWSIX-WASM userspace runtime does an XHR request to BROWSIX-SPEC to begin recording performance counter stats. (5) The benchmark harness finds the Chrome thread corresponding to the Web Worker `401.bzip2` process and attaches `perf` to the process. (6) At the end of the benchmark, the BROWSIX-WASM userspace runtime does a final XHR to the benchmark harness to end the `perf record` process. When the `runspec` program exits (after potentially invoking the test binary several times), the harness JS POSTs (7) a `tar` archive of the SPEC results directory to BROWSIX-SPEC. After BROWSIX-SPEC receives the full results archive, it unpacks the results to a temporary directory and validates the output using the `cmp` tool provided with SPEC 2006. Finally, BROWSIX-SPEC kills the browser process and records the benchmark results.

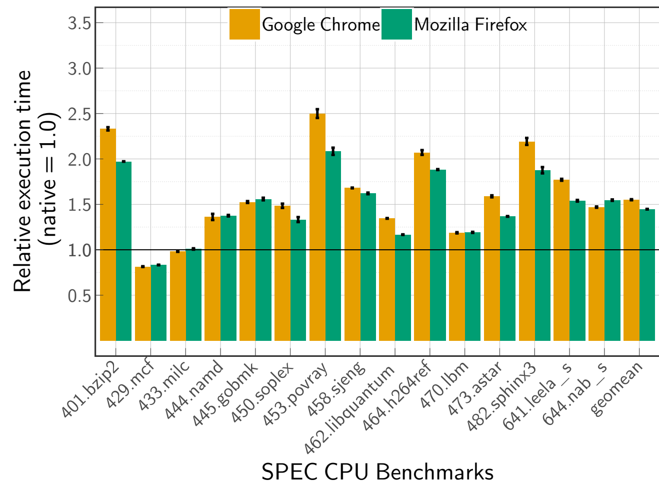
4 Evaluation

We use BROWSIX-WASM and BROWSIX-SPEC to evaluate the performance of WebAssembly using three benchmark

[2] `https://www.seleniumhq.org/`



(a)



(b)

Figure 3: The performance of the PolyBenchC and the SPEC CPU benchmarks compiled to WebAssembly (executed in Chrome and Firefox) relative to native, using BROWSIX-WASM and BROWSIX-SPEC. The SPEC CPU benchmarks exhibit higher overhead overall than the PolyBenchC suite, indicating a significant performance gap exists between WebAssembly and native.

suites: SPEC CPU2006, SPEC CPU2017, and PolyBenchC. We include PolybenchC benchmarks for comparison with the original WebAssembly paper [18], but argue that these benchmarks do not represent typical workloads. The SPEC benchmarks are representative and require BROWSIX-WASM to run successfully. We run all benchmarks on a 6-Core Intel Xeon E5-1650 v3 CPU with hyperthreading and 64 GB of RAM running Ubuntu 16.04 with Linux kernel v4.4.0. We run all benchmarks using two state-of-the-art browsers: Google Chrome 74.0 and Mozilla Firefox 66.0. We compile benchmarks to native code using Clang 4.0[3] and to WebAssembly using BROWSIX-WASM (which is based on Emscripten with Clang 4.0).[4] Each benchmark was executed five times. We report the average of all running times and the standard error. The execution time measured is the difference between wall clock time when the program starts, i.e. after WebAssembly JIT compilation concludes, and when the program ends.

4.1 PolyBenchC Benchmarks

Haas et al. [18] used PolybenchC to benchmark WebAssembly implementations because the PolybenchC benchmarks do not make system calls. As we have already argued, the PolybenchC benchmarks are small scientific kernels that are typically used to benchmark polyhedral optimization techniques, and do not represent larger applications. Nevertheless, it is still valuable for us to run PolybenchC with BROWSIX-WASM, because it demonstrates that our infrastructure for system calls does not have any overhead. Figure 3a shows the execution time of the PolyBenchC benchmarks in BROWSIX-WASM and when run natively. We are able to reproduce the majority of the results from the original WebAssembly paper [18]. We find that BROWSIX-WASM imposes a very low overhead: an average of 0.2% and a maximum of 1.2%.

4.2 SPEC Benchmarks

We now evaluate BROWSIX-WASM using the C/C++ benchmarks from SPEC CPU2006 and SPEC CPU2017 (the new C/C++ benchmarks and the speed benchmarks), which use system calls extensively. We exclude four data points that either do not compile to WebAssembly[5] or allocate more memory than WebAssembly allows.[6] Table 1 shows the absolute execution times of the SPEC benchmarks when running with BROWSIX-WASM in both Chrome and Firefox, and when running natively.

WebAssembly performs worse than native for all benchmarks except for `429.mcf` and `433.milc`. In Chrome, WebAssembly's maximum overhead is 2.5× over native and 7 out of 15 benchmarks have a running time within 1.5× of native. In Firefox, WebAssembly is within 2.08× of native and performs within 1.5× of native for 7 out of 15 benchmarks. On average, WebAssembly is 1.55× slower than native in Chrome, and 1.45× slower than native in Firefox. Table 2 shows the time required to compile the SPEC benchmarks

[3] The flags to Clang are `-O2 -fno-strict-aliasing`.

[4] BROWSIX-WASM runs Emscripten with the flags `-O2 -s TOTAL_MEMORY=1073741824 -s ALLOW_MEMORY_GROWTH=1 -fno-strict-aliasing`.

[5] `400.perlbench`, `403.gcc`, `471.omnetpp`, and `456.hmmer` from SPEC CPU2006 do not compile with Emscripten.

[6] From SPEC CPU2017, the `ref` dataset of `638.imagick_s` and `657.xz_s` require more than 4 GB RAM. However, these benchmarks do work with their `test` dataset.

Benchmark	Native	Google Chrome	Mozilla Firefox
401.bzip2	370 ± 0.6	864 ± 6.4	730 ± 1.3
429.mcf	221 ± 0.1	180 ± 0.9	184 ± 0.6
433.milc	375 ± 2.6	369 ± 0.5	378 ± 0.6
444.namd	271 ± 0.8	369 ± 9.1	373 ± 1.8
445.gobmk	352 ± 2.1	537 ± 0.8	549 ± 3.3
450.soplex	179 ± 3.7	265 ± 1.2	238 ± 0.5
453.povray	110 ± 1.9	275 ± 1.3	229 ± 1.5
458.sjeng	358 ± 1.4	602 ± 2.5	580 ± 2.0
462.libquantum	330 ± 0.8	444 ± 0.2	385 ± 0.8
464.h264ref	389 ± 0.7	807 ± 11.0	733 ± 2.4
470.lbm	209 ± 1.1	248 ± 0.3	249 ± 0.5
473.astar	299 ± 0.5	474 ± 3.5	408 ± 1.0
482.sphinx3	381 ± 7.1	834 ± 1.8	713 ± 3.6
641.leela_s	466 ± 2.7	825 ± 4.6	717 ± 1.2
644.nab_s	2476 ± 11	3639 ± 5.6	3829 ± 6.7
Slowdown: geomean	–	**1.55×**	**1.45×**
Slowdown: median	–	**1.53×**	**1.54×**

Table 1: Detailed breakdown of SPEC CPU benchmarks execution times (of 5 runs) for native (Clang) and WebAssembly (Chrome and Firefox); all times are in seconds. The median slowdown of WebAssembly is 1.53× for Chrome and 1.54× for Firefox.

using Clang and Chrome. (To the best of our knowledge, Firefox cannot report WebAssembly compile times.) In all cases, the compilation time is negligible compared to the execution time. However, the Clang compiler is orders of magnitude slower than the WebAssembly compiler. Finally, note that Clang compiles benchmarks from C++ source code, whereas Chrome compiles WebAssembly, which is a simpler format than C++.

4.2.1 BROWSIX-WASM Overhead

It is important to rule out the possibility that the slowdown that we report is due to poor performance in our implementation of BROWSIX-WASM. In particular, BROWSIX-WASM implements system calls without modifying the browser, and system calls involve copying data (§2), which may be costly. To quantify the overhead of BROWSIX-WASM, we instrumented its system calls to measure all time spent in BROWSIX-WASM. Figure 4 shows the percentage of time spent in BROWSIX-WASM in Firefox using the SPEC benchmarks. For 14 of the 15 benchmarks, the overhead is less than 0.5%. The maximum overhead is 1.2%. On average, the overhead of BROWSIX-WASM is only 0.2%. Therefore, we conclude that BROWSIX-WASM has negligible overhead and does not substantially affect the performance counter results of programs executed in WebAssembly.

Benchmark	Clang 4.0	Google Chrome
401.bzip2	1.9 ± 0.018	0.53 ± 0.005
429.mcf	0.3 ± 0.003	0.15 ± 0.005
433.milc	2.2 ± 0.02	0.3 ± 0.003
444.namd	4.6 ± 0.02	0.78 ± 0.004
445.gobmk	12.1 ± 0.2	1.4 ± 0.014
450.soplex	6.9 ± 0.01	1.2 ± 0.009
453.povray	15.3 ± 0.03	1.2 ± 0.012
458.sjeng	1.9 ± 0.01	0.35 ± 0.001
462.libquantum	6.9 ± 0.03	0.15 ± 0.002
464.h264ref	10.3 ± 0.06	1.0 ± 0.03
470.lbm	0.3 ± 0.001	0.14 ± 0.004
473.astar	0.73 ± 0.005	0.24 ± 0.004
482.sphinx3	3.0 ± 0.04	0.48 ± 0.007
641.leela_s	4.3 ± 0.05	0.74 ±0.003
644.nab_s	4.1 ± 0.03	0.41 ±0.001

Table 2: Compilation times of SPEC CPU benchmarks (average of 5 runs) for Clang 4.0 and WebAssembly (Chrome); all times are in seconds.

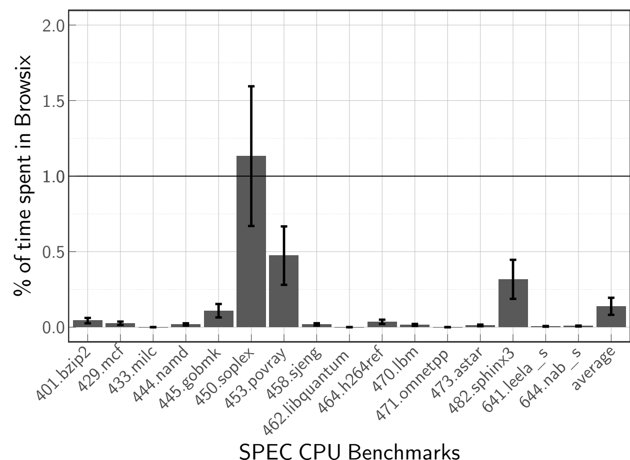


Figure 4: Time spent (in %) in BROWSIX-WASM calls in Firefox for SPEC benchmarks compiled to WebAssembly. BROWSIX-WASM imposes a mean overhead of only 0.2%.

4.2.2 Comparison of WebAssembly and `asm.js`

A key claim in the original work on WebAssembly was that it is significantly faster than `asm.js`. We now test that claim using the SPEC benchmarks. For this comparison, we modified BROWSIX-WASM to also support processes compiled to `asm.js`. The alternative would have been to benchmark the `asm.js` processes using the original BROWSIX. However, as we discussed earlier, BROWSIX has performance problems that would have been a threat to the validity of our results. Figure 5 shows the speedup of the SPEC benchmarks using WebAssembly, relative to their running time using `asm.js` using both Chrome and Firefox. WebAssembly outperforms `asm.js` in both browsers: the mean speedup is 1.54× in

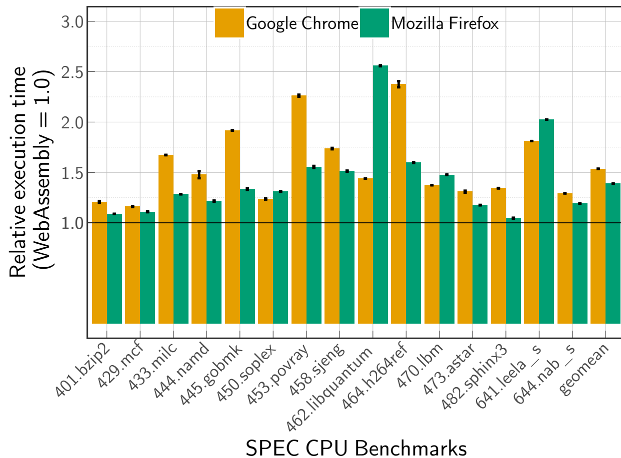


Figure 5: Relative time of `asm.js` to WebAssembly for Chrome and Firefox. WebAssembly is 1.54× faster than `asm.js` in Chrome and 1.39× faster than `asm.js` in Firefox.

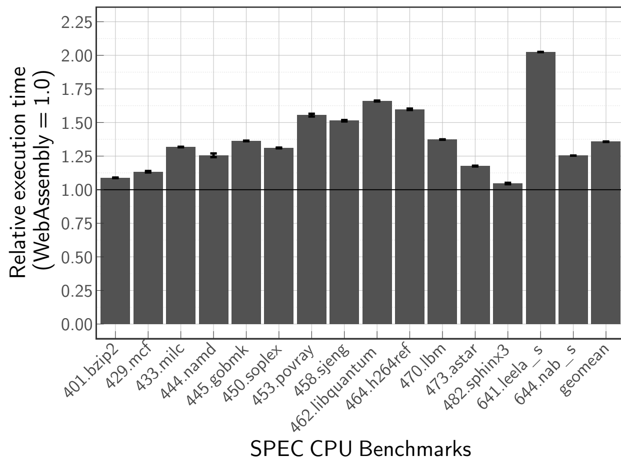


Figure 6: Relative *best* time of `asm.js` to the best time of WebAssembly. WebAssembly is 1.3× faster than `asm.js`.

Chrome and 1.39× in Firefox.

Since the performance difference between Chrome and Firefox is substantial, in Figure 6 we show the speedup of each benchmark by selecting the best-performing browser for WebAssembly and the best-performing browser of `asm.js` (i.e., they may be different browsers). These results show that WebAssembly consistently performs better than `asm.js`, with a mean speedup of 1.3×. Haas et al. [18] also found that WebAssembly gives a mean speedup of 1.3× over `asm.js` using PolyBenchC.

5 Case Study: Matrix Multiplication

In this section, we illustrate the performance differences between WebAssembly and native code using a C function that performs matrix multiplication, as shown in Figure 7a. Three matrices are provided as arguments to the function, and the results of `A` ($N_I \times N_K$) and `B` ($N_K \times N_J$) are stored in `C` ($N_I \times N_J$), where N_I, N_K, N_J are constants defined in the program.

In WebAssembly, this function is 2×–3.4× slower than native in both Chrome and Firefox with a variety of matrix sizes (Figure 8). We compiled the function with `-O2` and disabled automatic vectorization, since WebAssembly does not support vectorized instructions.

Figure 7b shows native code generated for the `matmul` function by `clang-4.0`. Arguments are passed to the function in the `rdi`, `rsi`, and `rdx` registers, as specified in the System V AMD64 ABI calling convention [9]. Lines 2 - 26 are the body of the first loop with iterator `i` stored in `r8d`. Lines 5 - 21 contain the body of the second loop with iterator `k` stored in `r9d`. Lines 10 - 16 comprise the body of the third loop with iterator `j` stored in `rcx`. Clang is able to eliminate a `cmp` instruction in the inner loop by initializing `rcx` with $-N_J$, incrementing `rcx` on each iteration at line 15, and using `jne` to test the zero flag of the status register, which is set to 1 when `rcx` becomes 0.

Figure 7c shows x86-64 code JITed by Chrome for the WebAssembly compiled version of `matmul`. This code has been modified slightly – `nop`s in the generated code have been removed for presentation. Function arguments are passed in the `rax`, `rcx`, and `rdx` registers, following Chrome's calling convention. At lines 1– 3, the contents of registers `rax`, `rdx`, and `rcx` are stored on the stack, due to registers spills at lines 7 - 9. Lines 7–45 are the body of the first loop with iterator `i` stored in `edi`. Lines 18–42 contain the body of second loop with iterator `k` stored in `r11`. Lines 27–39 are the body of the third loop with iterator `j` stored in `eax`. To avoid memory loads due to register spilling at lines 7– 9 in the first iteration of the first loop, an extra jump is generated at line 5. Similarly, extra jumps are generated for the second and third loops at line 16 and line 25 respectively.

5.1 Differences

The native code JITed by Chrome has more instructions, suffers from increased register pressure, and has extra branches compared to Clang-generated native code.

5.1.1 Increased Code Size

The number of instructions in the code generated by Chrome (Figure 7c) is 53, including `nop`s, while clang generated code (Figure 7b) consists of only 28 instructions. The poor instruction selection algorithm of Chrome is one of the reasons for increased code size.

Additionally, Chrome does not take advantage of all available memory addressing modes for x86 instructions. In Figure 7b Clang uses the `add` instruction at line 14 with register addressing mode, loading from and writing to a memory address in the same operation. Chrome on the other hand loads

```
1 void matmul (int C[NI][NJ],
2              int A[NI][NK],
3              int B[NK][NJ]) {
4   for (int i = 0; i < NI; i++) {
5     for (int k = 0; k < NK; k++) {
6       for (int j = 0; j < NJ; j++) {
7         C[i][j] += A[i][k] * B[k][j];
8       }
9     }
10  }
11 }
```

(a) `matmul` source code in C.

```
1 xor  r8d, r8d           #i <- 0
2 L1:                     #start first loop
3   mov  r10, rdx
4   xor  r9d, r9d         #k <- 0
5   L2:                   #start second loop
6     imul  rax, 4*NK, r8
7     add   rax, rsi
8     lea   r11, [rax + r9*4]
9     mov   rcx, -NJ     #j <- -NJ
10    L3:                 #start third loop
11      mov   eax, [r11]
12      mov   ebx, [r10 + rcx*4 + 4400]
13      imul  ebx, eax
14      add   [rdi + rcx*4 + 4*NJ], ebx
15      add   rcx, 1      #j <- j + 1
16    jne L3              #end third loop
17
18    add   r9,  1        #k <- k + 1
19    add   r10, 4*NK
20    cmp   r9,  NK
21  jne L2                #end second loop
22
23  add  r8,  1           #i <- i + 1
24  add  rdi, 4*NJ
25  cmp  r8,  NI
26 jne L1                 #end first loop
27 pop  rbx
28 ret
```

(b) Native x86-64 code for `matmul` generated by Clang.

```
1 mov [rbp-0x28],rax
2 mov [rbp-0x20],rdx
3 mov [rbp-0x18],rcx
4 xor edi,edi                  #i <- 0
5 jmp L1'
6 L1:                          #start first loop
7   mov ecx,[rbp-0x18]
8   mov edx,[rbp-0x20]
9   mov eax,[rbp-0x28]
10  L1':
11  imul r8d,edi,0x1130
12  add r8d,eax
13  imul r9d,edi,0x12c0
14  add r9d,edx
15  xor r11d,r11d              #k <- 0
16  jmp L2'
17  L2:                        #start second loop
18    mov ecx,[rbp-0x18]
19    L2':
20    imul r12d,r11d,0x1130
21    lea r14d,[r9+r11*4]
22    add r12d,ecx
23    xor esi,esi              #j <- 0
24    mov r15d,esi
25    jmp L3'
26    L3:                      #start third loop
27      mov r15d,eax
28      L3':
29      lea eax,[r15+0x1]      #j <- j + 1
30      lea edx,[r8+r15*4]
31      lea r15d,[r12+r15*4]
32      mov esi,[rbx+r14*1]
33      mov r15d,[rbx+r15*1]
34      imul r15d,esi
35      mov ecx,[rbx+rdx*1]
36      add ecx,r15d
37      mov [rbx+rdx*1],ecx
38      cmp eax,NJ             #j < NJ
39    jnz L3                   #end third loop
40  add r11,0x1                #k++
41  cmp r11d,NK                #k < NK
42  jnz L2                     #end second loop
43 add edi,0x1                 #i++
44 cmp edi,NI                  #i < NI
45 jnz L1                      #end first loop
46 retl
```

(c) x86-64 code JITed by Chrome from WebAssembly `matmul`.

Figure 7: Native code for `matmul` is shorter, has less register pressure, and fewer branches than the code JITed by Chrome. §6 shows that these inefficiencies are pervasive, reducing performance across the SPEC CPU benchmark suites.

the address in `ecx`, adds the operand to `ecx`, finally storing `ecx` at the address, requiring 3 instructions rather than one on lines 35–37.

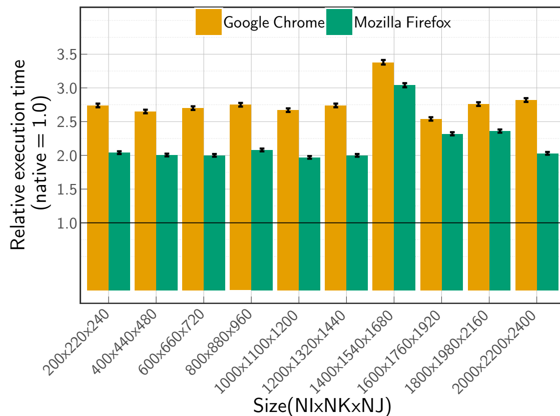


Figure 8: Performance of WebAssembly in Chrome and Firefox for different matrix sizes relative to native code. WebAssembly is always between 2× to 3.4× slower than native.

`perf` Event	Wasm Summary
`all-loads-retired (r81d0)` (Figure 9a) `all-stores-retired (r82d0)` (Figure 9b)	Increased register pressure
`branches-retired (r00c4)` (Figure 9c) `conditional-branches (r01c4)` (Figure 9d)	More branch statements
`instructions-retired (r1c0)` (Figure 9e) `cpu-cycles` (Figure 9f) `L1-icache-load-misses` (Figure 10)	Increased code size

Table 3: Performance counters highlight specific issues with WebAssembly code generation. When a raw PMU event descriptor is used, it is indicated by `rn`.

5.1.2 Increased Register Pressure

Code generated by Clang in Figure 7b does not generate any spills and uses only 10 registers. On the other hand, the code generated by Chrome (Figure 7c) uses 13 general purpose registers – all available registers (`r13` and `r10` are reserved by V8). As described in Section 5.1.1, eschewing the use of the register addressing mode of the `add` instruction requires the use of a temporary register. All of this register inefficiency compounds, introducing three register spills to the stack at lines 1–3. Values stored on the stack are loaded again into registers at lines 7–9 and line 18.

5.1.3 Extra Branches

Clang (Figure 7b) generates code with a single branch per loop by inverting the loop counter (line 15). In contrast, Chrome (Figure 7c) generates more straightforward code, which requires a conditional jump at the start of the loop. In addition, Chrome generates extra jumps to avoid memory loads due to register spills in the first iteration of a loop. For example, the jump at line 5 avoids the spills at lines 7– 9.

6 Performance Analysis

We use BROWSIX-SPEC to record measurements from all supported performance counters on our system for the SPEC CPU benchmarks compiled to WebAssembly and executed in Chrome and Firefox, and the SPEC CPU benchmarks compiled to native code (Section 3).

Table 3 lists the performance counters we use here, along with a summary of the impact of BROWSIX-WASM performance on these counters compared to native. We use these results to explain the performance overhead of WebAssembly over native code. Our analysis shows that the inefficiences described in Section 5 are pervasive and translate to reduced performance across the SPEC CPU benchmark suite.

6.1 Increased Register Pressure

This section focuses on two performance counters that show the effect of increased register pressure. Figure 9a presents the number of *load* instructions retired by WebAssembly-compiled SPEC benchmarks in Chrome and Firefox, relative to the number of load instructions retired in native code. Similarly, Figure 9b shows the number of *store* instructions retired. Note that a “retired” instruction is an instruction which leaves the instruction pipeline and its results are correct and visible in the architectural state (that is, not speculative).

Code generated by Chrome has 2.02× more load instructions retired and 2.30× more store instructions retired than native code. Code generated by Firefox has 1.92× more load instructions retired and 2.16× more store instructions retired than native code. These results show that the WebAssembly-compiled SPEC CPU benchmarks suffer from increased register pressure and thus increased memory references. Below, we outline the reasons for this increased register pressure.

6.1.1 Reserved Registers

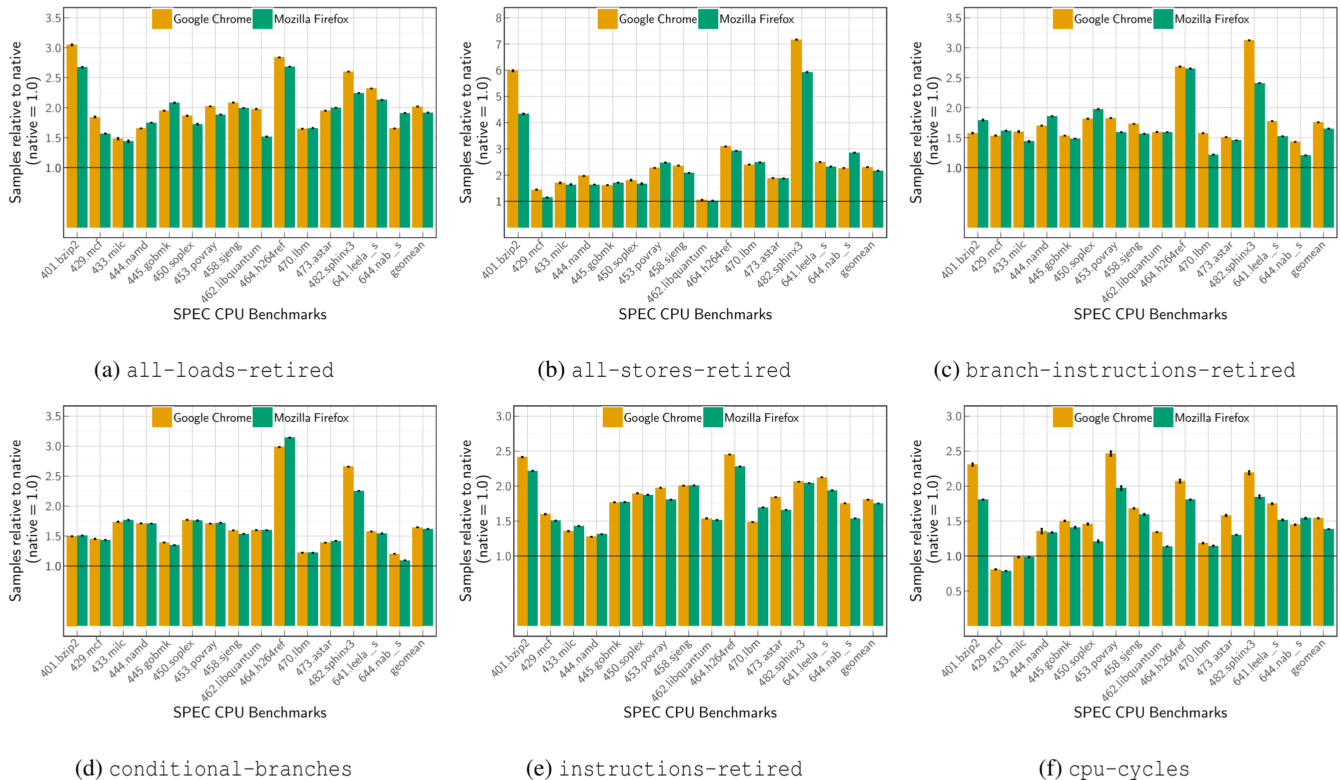
In Chrome, `matmul` generates three register spills but does not use two x86-64 registers: `r13` and `r10` (Figure 7c, lines 7– 9). This occurs because Chrome reserves these two registers.[7] For the JavaScript garbage collector, Chrome reserves `r13` to point to an array of GC roots at all times. In addition, Chrome uses `r10` and `xmm13` as dedicated scratch registers. Similarly, Firefox reserves `r15` as a pointer to the start of the heap, and `r11` and `xmm15` are JavaScript scratch registers.[8] None of these registers are available to WebAssembly code.

6.1.2 Poor Register Allocation

Beyond a reduced set of registers available to allocate, both Chrome and Firefox do a poor job of allocating the registers

[7] `https://github.com/v8/v8/blob/7.4.1/src/x64/register-x64.h`

[8] `https://hg.mozilla.org/mozilla-central/file/tip/js/src/jit/x64/Assembler-x64.h`



(a) `all-loads-retired` (b) `all-stores-retired` (c) `branch-instructions-retired`

(d) `conditional-branches` (e) `instructions-retired` (f) `cpu-cycles`

Figure 9: Processor performance counter samples for WebAssembly relative to native code.

they have. For example, the code generated by Chrome for `matmul` uses 12 registers while the native code generated by Clang only uses 10 registers (Section 5.1.2). This increased register usage—in both Firefox and Chrome—is because of their use of fast but not particularly effective register allocators. Chrome and Firefox both use a linear scan register allocator [36], while Clang uses a greedy graph-coloring register allocator [3], which consistently generates better code.

6.1.3 x86 Addressing Modes

The x86-64 instruction set offers several addressing modes for each operand, including a *register* mode, where the instruction reads data from register or writes data to a register, and memory address modes like *register indirect* or *direct offset* addressing, where the operand resides in a memory address and the instruction can read from or write to that address. A code generator could avoid unnecessary register pressure by using the latter modes. However, Chrome does not take advantage of these modes. For example, the code generated by Chrome for `matmul` does not use the register indirect addressing mode for the `add` instruction (Section 5.1.2), creating unnecessary register pressure.

6.2 Extra Branch Instructions

This section focuses on two performance counters that measure the number of branch instructions executed. Figure 9c shows the number of branch instructions retired by WebAssembly, relative to the number of branch instructions retired in native code. Similarly, Figure 9d shows the number of *conditional* branch instructions retired. In Chrome, there are 1.75× and 1.65× more unconditional and conditional branch instructions retired respectively, whereas in Firefox, there are 1.65× and 1.62× more retired. These results show that all the SPEC CPU benchmarks incur extra branches, and we explain why below.

6.2.1 Extra Jump Statements for Loops

As with `matmul` (Section 5.1.3), Chrome generates unnecessary jump statements for loops, leading to significantly more branch instructions than Firefox.

6.2.2 Stack Overflow Checks Per Function Call

A WebAssembly program tracks the current stack size with a global variable that it increases on every function call. The programmer can define the maximum stack size for the program. To ensure that a program does not overflow the stack,

both Chrome and Firefox add stack checks at the start of each function to detect if the current stack size is less than the maximum stack size. These checks includes extra comparison and conditional jump instructions, which must be executed on every function call.

6.2.3 Function Table Indexing Checks

WebAssembly dynamically checks all indirect calls to ensure that the target is a valid function and that the function's type at runtime is the same as the type specified at the call site. In a WebAssembly module, the function table stores the list of functions and their types, and the code generated by WebAssembly uses the function table to implement these checks. These checks are required when calling function pointers and virtual functions in C/C++. The checks lead to extra comparison and conditional jump instructions, which are executed before every indirect function call.

6.3 Increased Code Size

The code generated by Chrome and Firefox is considerably larger than the code generated by Clang. We use three performance counters to measure this effect. (i) Figure 9e shows the number of *instructions retired* by benchmarks compiled to WebAssembly and executed in Chrome and Firefox relative to the number of instructions retired in native code. Similarly, Figure 9f shows the relative number of *CPU cycles* spent by benchmarks compiled to WebAssembly, and Figure 10 shows the relative number of *L1 instruction cache load misses*.

Figure 9e shows that Chrome executes an average of 1.80× more instructions than native code and Firefox executes an average of 1.75× more instructions than native code. Due to poor instruction selection, a poor register allocator generating more register spills (Section 6.1), and extra branch statements (Section 6.2), the size of generated code for WebAssembly is greater than native code, leading to more instructions being executed. This increase in the number of instructions executed leads to increased L1 instruction cache misses in Figure 10. On average, Chrome suffers 2.83× more I-cache misses than native code, and Firefox suffers from 2.04× more L1 instruction cache misses than native code. More cache misses means that more CPU cycles are spent waiting for the instruction to be fetched.

We note one anomaly: although `429.mcf` has 1.6× more instructions retired in Chrome than native code and 1.5× more instructions retired in Firefox than native code, it runs *faster* than native code. Figure 3b shows that its slowdown relative to native is 0.81× in Chrome and 0.83× in Firefox. The reason for this anomaly is attributable directly to its lower number of L1 instruction cache misses. `429.mcf` contains a main loop and most of the instructions in the loop fit in the L1 instruction cache. Similarly, `433.milc` performance is better due to fewer L1 instruction cache misses. In `450.soplex` there are 4.6× more L1 instruction cache misses in Chrome and Firefox than native because of several virtual functions being executed, leading to more indirect function calls.

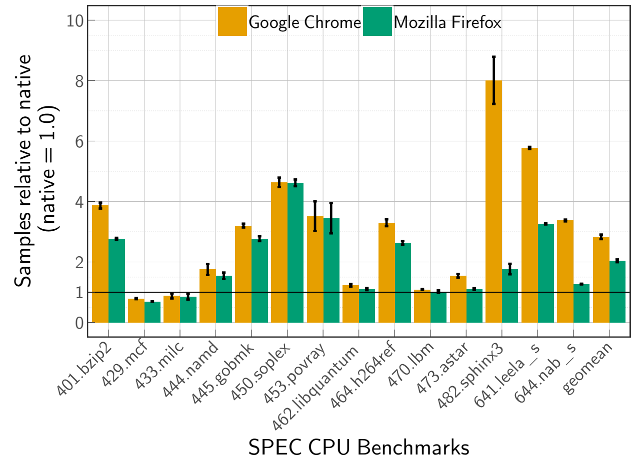


Figure 10: `L1-icache-load-misses` samples counted for SPEC CPU compiled to WebAssembly executed in Chrome and Firefox, relative to native. `458.sjeng` not shown in the graph exhibits 26.5× more L1 instruction cache misses in Chrome and 18.6× more in Firefox. The increased code size generated for WebAssembly leads to more instruction cache misses.

Performance Counter	Chrome	Firefox
`all-loads-retired`	2.02×	1.92×
`all-stores-retired`	2.30×	2.16×
`branch-instructions-retired`	1.75×	1.65×
`conditional-branches`	1.65×	1.62×
`instructions-retired`	1.80×	1.75×
`cpu-cycles`	1.54×	1.38×
`L1-icache-load-misses`	2.83×	2.04×

Table 4: The geomean of performance counter increases for the SPEC benchmarks using WebAssembly.

6.4 Discussion

It is worth asking if the performance issues identified here are fundamental. We believe that two of the identified issues are not: that is, they could be ameliorated by improved implementations. WebAssembly implementations today use register allocators (§6.1.2) and code generators (§6.2.1) that perform worse than Clang's counterparts. However, an offline compiler like Clang can spend considerably more time to generate better code, whereas WebAssembly compilers must be fast enough to run online. Therefore, solutions adopted by other JITs, such as further optimizing hot code, are likely applicable here [19, 32].

The four other issues that we have identified appear to

arise from the design constraints of WebAssembly: the stack overflow checks (§6.2.2), indirect call checks (§6.2.3), and reserved registers (§6.1.1) have a runtime cost and lead to increased code size (§6.3). Unfortunately, these checks are necessary for WebAssembly's safety guarantees. A redesigned WebAssembly, with richer types for memory and function pointers [23], might be able to perform some of these checks at compile time, but that could complicate the implementation of compilers that produce WebAssembly. Finally, a WebAssembly implementation in a browser must interoperate with a high-performance JavaScript implementation, which may impose its own constraints. For example, current JavaScript implementations reserve a few registers for their own use, which increases register pressure on WebAssembly.

7 Related Work

Precursors to WebAssembly There have been several attempts to execute native code in browsers, but none of them met all the design criteria of WebAssembly.

ActiveX [13] allows web pages to embed signed x86 libraries, however these binaries have unrestricted access to the Windows API. In contrast, WebAssembly modules are sandboxed. ActiveX is now a deprecated technology.

Native Client [11, 37] (NaCl) adds a module to a web application that contains platform specific machine code. NaCl introduced sandboxing techniques to execute platform specific machine code at near native speed. Since NaCl relies on static validation of machine code, it requires code generators to follow certain patterns, hence, supporting only a subset of the x86, ARM, and MIPS instructions sets in the browser. To address the inherent portability issue of NaCl, Portable NaCl (PNaCl) [14] uses LLVM Bitcode as a binary format. However, PNaCl does not provide significant improvement in compactness over NaCl and still exposes compiler and/or platform-specific details such as the call stack layout. Both have been deprecated in favor of WebAssembly.

`asm.js` is a subset of JavaScript designed to be compiled efficiently to native code. `asm.js` uses type coercions to avoid the dynamic type system of JavaScript. Since `asm.js` is a subset of JavaScript, adding all native features to `asm.js` such as 64-bit integers will first require extending JavaScript. Compared to `asm.js`, WebAssembly provides several improvements: (i) WebAssembly binaries are compact due to its lightweight representation compared to JavaScript source, (ii) WebAssembly is more straightforward to validate, (iii) WebAssembly provides formal guarantees of type safety and isolation, and (iv) WebAssembly has been shown to provide better performance than `asm.js`.

WebAssembly is a stack machine, which is similar to the Java Virtual Machine [21] and the Common Language Runtime [25]. However, WebAssembly is very different from these platforms. For example WebAssembly does not support objects and does not support unstructured control flow.

The WebAssembly specification defines its operational semantics and type system. This proof was mechanized using the Isabelle theorem prover, and that mechanization effort found and addressed a number of issues in the specification [35]. RockSalt [22] is a similar verification effort for NaCl. It implements the NaCl verification toolchain in Coq, along with a proof of correctness with respect to a model of the subset of x86 instructions that NaCl supports.

Analysis of SPEC Benchmarks using performance counters Several papers use performance counters to analyze the SPEC benchmarks. Panda et al. [26] analyze the SPEC CPU2017 benchmarks, applying statistical techniques to identify similarities among benchmarks. Phansalkar et al. perform a similar study on SPEC CPU2006 [27]. Limaye and Adegija identify workload differences between SPEC CPU2006 and SPEC CPU2017 [20].

8 Conclusions

This paper performs the first comprehensive performance analysis of WebAssembly. We develop BROWSIX-WASM, a significant extension of BROWSIX, and BROWSIX-SPEC, a harness that enables detailed performance analysis, to let us run the SPEC CPU2006 and CPU2017 benchmarks as WebAssembly in Chrome and Firefox. We find that the mean slowdown of WebAssembly vs. native across SPEC benchmarks is 1.55× for Chrome and 1.45× for Firefox, with peak slowdowns of 2.5× in Chrome and 2.08× in Firefox. We identify the causes of these performance gaps, providing actionable guidance for future optimization efforts.

Acknowledgements We thank the reviewers and our shepherd, Eric Eide, for their constructive feedback. This work was partially supported by NSF grants 1439008 and 1413985.

References

[1] Blazor. https://blazor.net/. [Online; accessed 5-January-2019].

[2] Compiling from Rust to WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm. [Online; accessed 5-January-2019].

[3] LLVM Reference Manual. https://llvm.org/docs/CodeGenerator.html.

[4] NaCl and PNaCl. https://developer.chrome.com/native-client/nacl-and-pnacl. [Online; accessed 5-January-2019].

[5] PolyBenchC: the polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/. [Online; accessed 14-March-2017].

[6] Raise Chrome JS heap limit? - Stack Overflow. https://stackoverflow.com/questions/43643406/raise-chrome-js-heap-limit. [Online; accessed 5-January-2019].

[7] Use cases. https://webassembly.org/docs/use-cases/.

[8] WebAssembly. https://webassembly.org/. [Online; accessed 5-January-2019].

[9] System V Application Binary Interface AMD64 Architecture Processor Supplement. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[10] Steve Akinyemi. A curated list of languages that compile directly to or have their VMs in WebAssembly. https://github.com/appcypher/awesome-wasm-langs. [Online; accessed 5-January-2019].

[11] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366. ACM, 2011.

[12] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725. ACM, 2010.

[13] David A Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[14] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable Native Client Executables. https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf, 2010.

[15] Brendan Eich. From ASM.JS to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015. [Online; accessed 5-January-2019].

[16] Eric Elliott. What is WebAssembly? https://tinyurl.com/o5h6daj, 2015. [Online; accessed 5-January-2019].

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478. ACM, 2009.

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200. ACM, 2017.

[19] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, 2008.

[20] Ankur Limaye and Tosiron Adegbija. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[22] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404. ACM, 2012.

[23] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[24] Richard Musiol. A compiler from Go to JavaScript for running Go code in a browser. `https://github.com/gopherjs/gopherjs`, 2016. [Online; accessed 5-January-2019].

[25] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228. Springer, 2002.

[26] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2018.

[27] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 412–423. ACM, 2007.

[28] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Unix in your browser tab. `https://browsix.org`.

[29] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the Gap Between Unix and the Browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 253–266. ACM, 2017.

[30] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12. ACM, 2010.

[31] Marija Selakovic and Michael Pradel. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72. ACM, 2016.

[32] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-in-time Compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 180–195. ACM, 2001.

[33] Luke Wagner. asm.js in Firefox Nightly | Luke Wagner's Blog. `https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/`. [Online; accessed 21-May-2019].

[34] Luke Wagner. A WebAssembly Milestone: Experimental Support in Multiple Browsers. `https://hacks.mozilla.org/2016/03/a-webassembly-milestone/`, 2016. [Online; accessed 5-January-2019].

[35] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65. ACM, 2018.

[36] Christian Wimmer and Michael Franz. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 170–179. ACM, 2010.

[37] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 2009.

[38] Alon Zakai. asm.js. `http://asmjs.org/`. [Online; accessed 5-January-2019].

[39] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312. ACM, 2011.

Extension Framework for File Systems in User space

Ashish Bijlani
Georgia Institute of Technology

Umakishore Ramachandran
Georgia Institute of Technology

Abstract

User file systems offer numerous advantages over their in-kernel implementations, such as ease of development and better system reliability. However, they incur heavy performance penalty. We observe that existing user file system frameworks are highly general; they consist of a minimal interposition layer in the kernel that simply forwards all low-level requests to user space. While this design offers flexibility, it also severely degrades performance due to frequent kernel-user context switching.

This work introduces EXTFUSE, a framework for developing extensible user file systems that also allows applications to register "thin" specialized request handlers in the kernel to meet their specific operative needs, while retaining the complex functionality in user space. Our evaluation with two FUSE file systems shows that EXTFUSE can improve the performance of user file systems with less than a few hundred lines on average. EXTFUSE is available on GitHub.

1 Introduction

User file systems not only offer better security (i.e., unprivileged execution) and reliability [46] when compared to in-kernel implementations, but also ease the development and maintenance/debugging processes. Therefore, many approaches to develop user space file systems have also been proposed for monolithic Operating Systems (OS), such as Linux and FreeBSD. While some approaches target specific systems [25, 45, 53], a number of general-purpose frameworks for implementing user file systems also exist [3, 14, 29, 33, 47]. FUSE [47], in particular, is the state-of-the-art framework for developing user file systems. Over a hundred FUSE file system have been created in academic/research [11, 32, 37, 41, 44, 49], as well as in production settings [9, 24, 40, 52].

Being general-purpose, the primary goal of the aforementioned frameworks is to enable easy, yet fully-functional implementation of file systems in user space supporting multiple different functionalities. To do so, they implement a minimal kernel driver that interfaces with the Virtual File System (VFS) operations and simply forwards all low-level requests to user space. For example, when an application (app) makes an `open()` system call, the VFS issues a `lookup` request for each path component. Similarly, `getxattr` requests are issued to read security labels while serving `write()` system calls. Such low-level requests are simply forwarded to user space. This design offers flexibility to developers to easily implement their functionality and apply custom optimizations, but also incurs a high overhead due to frequent user-kernel switching and data copying. For example, despite several recent optimizations, even a simple passthrough FUSE file system can introduce up to 83% overhead on an SSD [50]. As a result, some FUSE file systems have been replaced by alternative implementations in production [8, 22, 24].

There have been attempts to address performance issues of user file system frameworks, for example, by eliminating user-kernel switching in FUSE under certain scenarios [28, 34]. Nevertheless, the optimizations proposed pertain to their specific use cases and do not address the inherent design limitations of existing frameworks.

We observe that the interfaces exported by existing user file system frameworks are too low-level and general-purpose. As such, they fail to match the specific operative needs of file systems. For example, `getxattr` requests during `write()` can be completely eliminated for files containing no security labels. `lookup` replies from the daemon could be cached and validated in the kernel to reduce context switches to user space. Similarly, when stacking sandboxing functionality for enforcing custom permissions checks in `open()` system call, I/O requests (e.g., `read/write`) could be passed directly through the host file system. Nevertheless, modifying existing frameworks to efficiently address specific functional and performance requirements of each use case is impractical.

We borrow the idea of safely extending system services at runtime from past works [6, 13, 43, 56] and propose to address the performance issues in existing user file systems frameworks by allowing developers to safely extend the functionality of the kernel driver at runtime for specialized handling of their use case. This work introduces EXTFUSE, an extension framework for file systems in user space that allows developers to define specialized "thin" *extensions* along with auxiliary data structures for handling low-level requests in the kernel. The extensions are safely executed under a sandboxed runtime environment in the kernel immediately as the requests are issued from the upper file system layer (e.g., VFS), thereby offering a fine-grained ability to either serve each request entirely in the kernel (*fast path*) or fall back to the existing complex logic in user space (*slow path*) to achieve the desired operative goals of functionality and performance. The fast and slow paths can access (and modify) the auxiliary data structures to define custom logic for handling requests.

ExtFUSE consists of three components. First, a helper user library that provides a familiar set of file system APIs to register extensions and implement custom fast-path functionality in a subset of the C language. Second, a wrapper (no-ops) interposition driver that bridges with the low-level VFS interfaces and provides the necessary support to forward requests to the registered kernel extensions as well as to the lower file system, as needed. Third, an in-kernel Virtual Machine (VM) runtime that safely executes the extensions.

We have built ExtFUSE to work in concert with existing user file system frameworks to allow both the fast and the existing slow path to coexist with no overhauling changes to the design of the target user file system. Although there are several user file system frameworks, this work focuses only on FUSE because of its wide-spread use. Nonetheless, we have implemented ExtFUSE as in a modular fashion so it can be easily adopted for others.

We added support for ExtFUSE in four popular FUSE file systems, namely LoggedFS [16], Android sdcard daemon, MergerFS, and BindFS [35]. Our evaluation of the first two shows that ExtFUSE can offer substantial performance improvements of user file systems by adding less than a few hundred lines, on average.

This paper makes the following contributions:

- We identify optimization opportunities in user file system frameworks for monolithic OSes (§2) and propose extensible user file systems.
- We present the design (§3) and architecture (§3.4) of ExtFUSE, an extension framework for user file systems that offers the performance of kernel file systems, while retaining the safety properties of user file systems.
- We demonstrate the applicability of ExtFUSE by adopting it for FUSE, the state-of-the-art user file system framework, and evaluating it on Linux (§6).
- We show the practical benefits and limitations of the ExtFUSE with two popular FUSE file systems, one deployed in production (§6.2).

2 Background and Extended Motivation

This section provides a brief technical background on FUSE and its limitations that motivate our work.

2.1 FUSE

FUSE is the state-of-the-art framework for developing user file systems. It consists of a loadable kernel driver and a helper user-space library that provides a set of portable APIs to allow users to implement their own fully-functional file system as an unprivileged daemon process on Unix-based systems with no additional kernel support. The driver is a simple interposition layer that only serves as a communication channel between the user-space daemon and the VFS. It registers a new file system to interface with the VFS operations and directly forwards all low-level requests to the daemon, as received.

The library provides two different sets of APIs. First, a `fuse_lowlevel_ops` interface that exports all VFS operations such as `lookup` for path to inode mapping. It is used by file systems that need to access low-level abstractions (e.g., inodes) for custom optimizations. Second, a high-level `fuse_operations` interface that builds on the low-level APIs. It hides complex abstractions and offers a simple (e.g., path-based) API for the ease of development. Depending on their particular use case, developers can adopt either of the two APIs. Furthermore, many operations in both APIs are optional. For example, developers can choose to not handle extended attributes (xattrs) operations (e.g., `getxattr`).

As apps make system calls, the VFS layer forwards all low-level requests to the kernel driver. For example, to serve the `open()` system call on Linux, the VFS issues multiple `lookup` requests to the driver, one for each input path component. Similarly, every `write()` request is preceded by a `getxattr` request from the VFS to fetch the security labels. The driver queues up all requests, along with the relevant parameters, for the daemon through `/dev/fuse` device file and blocks the calling app thread until the requests are served. The daemon, through the `libfuse` interface, retrieves the requests from the queue, processes them as needed, and enqueues the results for the driver to read. The driver copies the results, populating VFS caches, as appropriate (e.g., page cache for `read`/`write`, dcache for dir entries from `lookup`), and wakes the app thread and returns the results to it. Once cached, the subsequent accesses to the same data are served with no user-kernel round-trip communication.

In addition to the common benefits of user file systems such as ease of development and maintenance, FUSE also provides app-transparency and fine-grained control to developers over the low-level API to easily support their custom functionality. Furthermore on Linux, the FUSE driver is GPL-licensed, which detaches the implementation in user space from legal obligations [57]. It also supports multiple language bindings (e.g., Python, Go, etc.), thereby enabling access to ecosystem of third-party libraries for reuse.

Given its general-purpose design and numerous advantages, over a hundred FUSE file systems with different functionalities have been created. A large majority of them are stackable; that is, they introduce incremental functionality on the host file system [38]. FUSE has long served as a popular tool for quick experimentation and prototyping new file systems in academic and research settings [11, 32, 37, 41, 44, 49]. However, more recently a host of FUSE file systems have also been deployed in production. Both Gluster [40] and Ceph [52] cluster file systems use a FUSE network client implementation. Android v4.4 introduced FUSE sdcard daemon (stackable) to add multi-user support and emulate FAT functionality on the EXT4 file system [24].

While exporting low-level abstractions and VFS interfaces offers more control and flexibility to developers, this design comes with a cost. It induces frequent user-kernel round-

Media	W/ xattr (%Diff)		W/O xattr (%Diff)	
	SW	RW	SW	RW
HDD	-0.29	-3.80	-0.17	-2.83
SSD	-11.5	-23.38	+0.31	-12.24

Table 1: Percentage difference between I/O throughput (ops/sec) for EXT4 vs FUSE StackfsOpt (see §6) under single thread 4K Seq Write (SW), and Rand Write (RW) settings on a 60GB file across different storage media as reported by Filebench [48]. Received 15 million getxattr requests.

trip communication and data copying, and thus inevitably yields poor runtime performance. Nonetheless, FUSE has evolved significantly over the years; several optimizations have been added to minimize the user-kernel communication: zero-copy data transfer (splicing), and utilizing system-wide shared VFS caches (page cache for data I/O and dentry cache for metadata). However, despite these optimizations, FUSE severely degrades runtime performance in a host of scenarios. For instance, data caching improves I/O throughput by batching requests (e.g., read-aheads, small writes), but it does not help apps that perform random reads or demand low write latency (e.g., databases). Splicing is only used for over 4K requests. Therefore, apps with small writes/reads (e.g., Android apps, etc.) will incur data copying overheads. Worse yet, the overhead is higher with faster storage media. Even a simple passthrough (no-ops) FUSE file system can introduce up to 83% overhead for metadata-heavy workloads on SSD [50].

The performance penalty incurred by user file systems overshadows their benefits. Consequently, some user file systems have been replaced with alternative implementations in production. For instance, Android v7.0 replaced the sdcard daemon with its in-kernel implementation after several years [24]. Ceph [52] adopted an in-kernel client for Linux kernel [8].

2.2 Generality vs Specialization

We observe that being highly general-purpose, the FUSE framework induces unnecessary user-kernel communication in many cases, yielding low throughput and high latency. For instance, `getxattr` requests generated by the VFS during `write()` system calls (one per `write`) can double the number of user-kernel transitions, which decreases the I/O throughput of sequential writes by over 27% and random writes by over 44% compared to native (EXT4) performance (Table 1). Moreover, the penalty incurred is higher with the faster media.

Nevertheless, simple filtering or caching metadata replies in the kernel can substantially reduce user-kernel communication. For instance, by caching the last `getxattr` reply in the FUSE driver and simply validating the cached state for every subsequent `getxattr` request from the VFS on the same file, unnecessary user-kernel transitions can be eliminated to achieve significant improvement in `write` performance. Similarly, replies from other metadata operations, such as `lookup` and `getattr` can be cached, validated, and served entirely within the kernel. Note that custom validation of cached metadata is imperative, and lack of support to do so may result in incorrect behavior as happens in the case of optimized FUSE that leverages VFS caches to serve requests (§5.1).

Many stackable user file systems add a thin layer of functionality; they perform simple checks in a few operations and pass remaining requests directly through the host (lower) file system. LoggedFS [16] filters requests that must be logged and do so by accessing host file system services. Union file systems such as MergerFS [20] determine the backend host file in `open()` and redirects I/O requests to it. Android sdcard daemon performs access permission checks only in metadata operations (e.g., `open`, `lookup`), but data I/O requests (e.g., `read`, `write`, etc.) are simply forwarded to the lower file system. Thin functionality that realizes such use cases does not need any complex processing in user space, and therefore can easily be stacked in the kernel, thereby avoiding expensive user-kernel switching to yield lower latency and higher throughput for the same functionality. Furthermore, data I/O requests could be directly forwarded to the host file system.

The FUSE framework offers a few configuration (config) options to the developers to tune the behavior of its kernel driver for their use case. However, those options are coarse-grained and implemented as fixed checks embedded in the driver code; thus, it offers limited static control. For example, for file systems that do not support certain operations (e.g., `getxattr`), the FUSE driver caches this knowledge upon first `ENOSUPPORT` reply and does not issue such requests subsequently. While this benefits the file systems that completely omit certain functionality (e.g., security xattr labels), it does not allow custom and fine-grained filtering of requests that may be desired by some file systems supporting only partial functionality. For example, file systems providing encryption (or compression) functionality may desire a fine-grained custom control over the kernel driver to skip the decryption (or decompression) operation in user space during `read()` requests on non-sensitive (or unzipped) files.

As such, the FUSE framework proves to be too low-level and general-purpose in many cases. While it enables a number of different use cases, modifying the framework to efficiently handle special needs of each use case is impractical. This is a typical unbalanced generality vs. specialization problem, which can be addressed by extending the functionality of the FUSE driver in the kernel [6, 13, 43].

3 Design

In this section, we 1) present an overview of EXTFUSE, 2) discuss the design goals and challenges we faced, and 3) mechanisms we adopted to address those challenges.

3.1 Overview

EXTFUSE is a framework for developing extensible FUSE file systems for UNIX-like monolithic OSes. It allows the

unprivileged FUSE daemon processes to register "thin" extensions in the kernel for specialized handling of low-level file system requests, while retaining their existing complex logic in user space to achieve the desired level of performance.

The registered extensions are safely executed under a sandboxed eBPF runtime environment in the kernel (§3.3), immediately as requests are issued from the upper file system (e.g., VFS). Sandboxing enables the FUSE daemon to safely extend the functionality of the driver at runtime and offers a fine-grained ability to either serve each request entirely in the kernel or fall back to user space, thereby offering safety of user space and performance of kernel file systems.

EXTFUSE also provides a set of APIs to create shared key-value data structures (called maps) that can host abstract data blobs. The user-space daemon and its kernel extensions can leverage maps to store/manipulate their custom data types as needed to work in concert and serve file system requests in the kernel without incurring expensive user-kernel round-trip communication if deemed unnecessary.

3.2 Goals and Challenges

The over-arching goal of EXTFUSE is to carefully balance the safety and runtime extensibility of user file systems to achieve the desired level of performance and specialized functionality. Nevertheless, in order for developers to use EXTFUSE, it must also be easy to adopt. We identify the following concrete design goals.

Design Compatibility. The abstractions and interfaces offered by EXTFUSE framework must be compatible with FUSE without hindering existing functionality or properties. It must be general-purpose so as to support multiple different use cases. Developers must further be able to adopt EXTFUSE for their use case without overhauling changes to their existing design. It must be easy for them to implement specialized extensions with little to no knowledge of the underlying implementation details.

Modular Extensibility. EXTFUSE must be highly modular and limit any unnecessary new changes to FUSE. Particularly, developers must be able to retain their existing user-space logic and introduce specialized extensions only if needed.

Balancing Safety and Performance. Finally with EXTFUSE, even unprivileged (and untrusted) FUSE daemon processes must be able to safely extend the functionality of the driver as needed to offer performance that is as close as possible to the in-kernel implementation. However, unlike Microkernels [2, 23, 30] that host system services in separate protection domains as user processes or the OSes that have been developed with safe runtime extensibility as a design goal [6, 13], extending system services of general-purpose UNIX-like monolithic OSes poses a design trade-off question between the safety and performance requirements.

Untrusted extensions must be as lightweight as possible, with their access restricted to only a few well-defined APIs to guarantee safety. For example, kernel file systems offer near-native performance, but executing complex logic in the kernel results in questionable reliability. Additionally, most OS kernels employ Address Space Randomization [36], Data Execution Prevention [5], etc. for code protection and hiding memory pointers. Providing unrestricted kernel access to extensions can render such protections useless. Therefore, extensions must not be able to access arbitrary memory addresses or leak pointer values to user space.

However, severely restricting extensibility can prevent user file systems from fully meeting their operative performance and functionality goals, thus defeating the purpose of extensions in the first place. Therefore, EXTFUSE must carefully balance safety and performance goals.

Correctness. Furthermore, specialized extensions can alter the existing design of user file systems, which can lead to correctness issues. For example, there will be two separate paths (fast and slow) both operating on the requests and data structs at the same time. The framework must provide a way for them to synchronize and offer safe concurrent accesses.

3.3 eBPF

EXTFUSE leverages extended BPF (eBPF) [26], an in-kernel Virtual Machine (VM) runtime framework to load and safely execute user file system extensions.

Richer functionality. eBPF is an extension of classic Berkeley Packet Filters (BPF), an in-kernel interpreter for a pseudo machine architecture designed to only accept simple network filtering rules from user space. It enhances BPF to include more versatility, such as 64-bit support, a richer instruction set (e.g., call, cond jump), more registers, and native performance through JIT compilation.

High-level language support. The eBPF bytecode backend is also supported by Clang/LLVM compiler toolchain, which allows functionality logic to be written in a familiar high-level language, such as C and Go.

Safety. The eBPF framework provides a safe execution environment in the kernel. It prohibits execution of arbitrary code and access to arbitrary kernel memory regions; instead, the framework restricts access to a set of kernel helper APIs depending on the target kernel subsystem (e.g., network) and required functionality (e.g., packet handling). The framework includes a static analyzer (called verifier) that checks the correctness of the bytecode by performing an exhaustive depth-first search through its control flow graph to detect problems, such as infinite loops, out-of-bound, and illegal memory errors. The framework can also be configured to allow or deny eBPF bytecode execution request from unprivileged processes.

Key-Value Maps. eBPF allows user space to create *map* data structures to store arbitrary key-value blobs using system calls and access them using file descriptors. Maps are also accessible to eBPF bytecode in the kernel, thus providing a communication channel between user space and the bytecode

to define custom key-value types and share execution state or data. Concurrent accesses to maps are protected under read-copy update (RCU) synchronization mechanism. However, maps consume unswappable kernel memory. Furthermore, they are either accessible to everyone (e.g., by passing file descriptors) or only to `CAP_SYS_ADMIN` processes.

eBPF is a part of the Linux kernel and is already used heavily by networking, tracing, and profiling subsystems. Given its rich functionality and safety properties, we adopt eBPF for providing support for extensible user file systems. Specifically, we define a white-list of kernel APIs (including their parameters and return types), and abstractions that user file system extensions can safely use to realize their specialized functionality. The eBPF verifier utilizes the whitelist to validate the correctness of the extensions. We also build on eBPF abstractions (e.g., maps) and apply further access restrictions to enable safe in-kernel execution, as needed.

3.4 Architecture

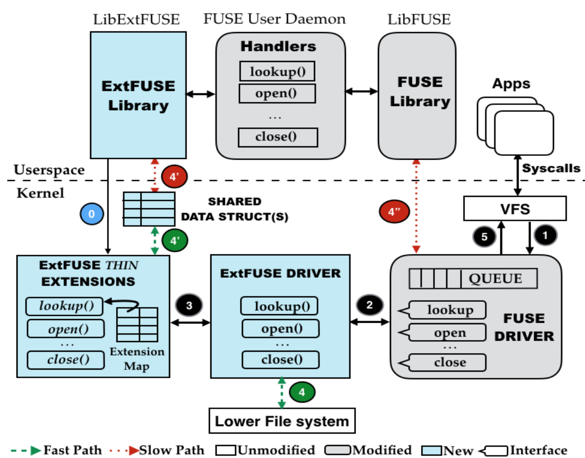


Figure 1: Architectural view of the EXTFUSE framework. The components modified or introduced have been highlighted.

Figure 1 shows the architecture of the EXTFUSE framework. It is enabled by three core components, namely a kernel file system (`driver`), a user library (`libExtFUSE`), and an in-kernel eBPF virtual machine runtime (`VM`).

The EXTFUSE driver uses interposition technique to interface with FUSE at low-level file system operations. However, unlike the FUSE driver that simply forwards file system requests to user space, the EXTFUSE driver is capable of directly delivering requests to in-kernel handlers (extensions). It can also forward a few restricted set of requests (e.g., `read`, `write`) to the host (lower) file system, if present. The latter is needed for stackable user file systems that add thin functionality on top of the host file system. `libExtFUSE` exports a set of APIs and abstractions for serving requests in the kernel, hiding the underlying implementation details.

Use of `libExtFUSE` is optional and independent of `libfuse`. The existing file system handlers registered with `libfuse` continue to reside in user space. Therefore, their invocation incurs context switches, and thus, we refer to their execution as the *slow* path. With EXTFUSE, user space can also register kernel extensions that are invoked immediately as file system requests are received from the VFS in order to allow serving them in the kernel. We refer to the in-kernel execution as the *fast* path. Depending upon the return values from the fast path, the requests can be marked as served or be sent to the user-space daemon via the slow path to avail any complex processing as needed. Fast path can also return a special value that instructs the EXTFUSE driver to interpose and forward the request to the lower file system. However, this feature is only available to stackable user file systems and is verified when the extensions are loaded in the kernel.

The fast path interfaces exported by `libExtFUSE` are the same as those exported by `libfuse` to the slow path. This is important for easy transfer of design and portability. We leverage eBPF support in the LLVM/Clang compiler toolchain to provide developers with a familiar set of APIs and allow them to implement their custom functionality logic in a subset of the C language.

The extensions are loaded and executed inside the kernel under the eBPF VM sandbox, thereby providing user space a fine-grained ability to safely extend the functionality of FUSE kernel driver at runtime for specialized handling of each file system request.

FS Interface	API(s)	Abstractions	Description
Low-level	`fuse_lowlevel_ops`	`Inode`	FS Ops
Kernel Access	**API(s)**	**Abstractions**	**Description**
eBPF Funcs	`bpf_*`	`UID, PID, etc.`	Helper Funcs
FUSE	`extfuse_reply_*`	`fuse_reply_*`	Req Output
Kernel	`bpf_set_pasthru`	`FileDesc`	Enable Pthru
Kernel	`bpf_clear_pasthru`	`FileDesc`	Disable Pthru
DataStructs	**API(s)**	**Abstractions**	**Description**
SHashMap	`CRUD`	`Key/Val`	Hosts arbitrary data blobs
InodeMap	`CRUD`	`FileDesc`	Hosts upper-lower inode pairs

Table 2: APIs and abstractions provided by EXTFUSE. It provides FUSE-like file system interface for easy portability. CRUD (create, read, update, and delete) APIs are offered for map data structures to operate on Key/Value pairs. Kernel accesses are restricted to standard eBPF kernel helper functions. We introduced APIs to access the same FUSE request parameters as available to user space.

3.5 EXTFUSE APIs and Abstractions

`libExtFUSE` provides a set of high-level APIs and abstractions to the developers for easy implementation of their specialized extensions, hiding the complex implementation details. Table 2 summarizes the APIs. For handling file system operations, `libExtFUSE` exports the familiar set of FUSE interfaces and corresponding abstractions (e.g., inode) for design compatibility. Both low-level as well as high-level file system interfaces are available, offering flexibility and development ease. Furthermore, as with `libfuse`, the daemon can reg-

ister extensions for a few or all of the file system APIs, offering them flexibility to implement their functionality with no additional development burden. The extensions receive the same request parameters (`struct fuse_[in,out]`) as the user-space daemon. This design choice not only conforms to the principle of least privilege, but also offers the user-space daemon and the extensions the same interface for easy portability.

For hosting/sharing data between the user daemon and kernel extensions, `libExtFUSE` provides a secure variant of eBPF `HashMap` key/value data structure called `SHashMap` that stores arbitrary key/value blobs. Unlike regular eBPF maps that are either accessible to all user processes or only to `CAP_SYS_ADMIN` processes, `SHashMap` is only accessible by the unprivileged daemon that creates it. `libExtFUSE` further abstracts low-level details of `SHashMap` and provides high-level `CRUD` APIs to create, read, update, and delete entries (key/value pairs).

EXTFUSE also provides a special `InodeMap` to enable passthrough I/O feature for stackable EXTFUSE file systems (§5.2). Unlike `SHashMap` that stores arbitrary entries, `InodeMap` takes open file handle as key and stores a pointer to the corresponding lower (host) inode as value. Furthermore, to prevent leakage of inode object to user space, the `InodeMap` values can only be read by the EXTFUSE driver.

3.6 Workflow

To understand how EXTFUSE facilitates implementation of extensible user file systems, we describe its workflow in detail. Upon mounting the user file system, FUSE driver sends `FUSE_INIT` request to the user-space daemon. At this point, the user daemon checks if the OS kernel supports EXTFUSE framework by looking for `FUSE_CAP_ExtFUSE` flag in the request parameters. If supported, the daemon must invoke `libExtFUSE init` API to load the eBPF program that contains specialized handlers (extensions) into the kernel and register them with the EXTFUSE driver. This is achieved using `bpf_load_prog` system call, which invokes eBPF verifier to check the integrity of the extensions. If failed, the program is discarded and the user-space daemon is notified of the errors. The daemon can then either exit or continue with default FUSE functionality. If the verification step succeeds and the JIT engine is enabled, the extensions are processed by the JIT compiler to generate machine assembly code ready for execution, as needed.

Extensions are installed in a `bpf_prog_type` map (called *extension map)*, which serves effectively as a jump table. To invoke an extension, the FUSE driver simply executes a `bpf_tail_call` (far jump) with the FUSE operation code (e.g., `FUSE_OPEN`) as an index into the extension map. Once the eBPF program is loaded, the daemon must inform EXTFUSE driver about the kernel extensions by replying to `FUSE_INIT` containing identifiers to the extension map.

Once notified, EXTFUSE can safely load and execute the extensions at runtime under the eBPF VM environment. Every request is first delivered to the fast path, which may decide to 1) serve it (e.g., using data shared between the fast and slow paths), 2) pass the request through to the lower file system (e.g., after modifying parameters or performing access checks), or 3) take the slow path and deliver the request to user space for complex processing logic (e.g., data encryption), as needed. Since the execution path is chosen per-request independently and the fast path is always invoked first, the kernel extensions and user daemon can work in concert and synchronize access to requests and shared data structures. It is important to note that the EXTFUSE driver only acts as a thin interposition layer between the FUSE driver and kernel extensions, and in some cases, between the FUSE driver and the lower file system. As such, it does not perform any I/O operation or attempts to serve requests on its own.

Component	Version	Loc Modified	Loc New
FUSE kernel driver	4.11.0	312	874
FUSE user-space library	3.2.0	23	84
EXTFUSE user-space library	-	-	581

Table 3: Changes made to the existing Linux FUSE framework to support EXTFUSE functionality.

4 Implementation

To implement EXTFUSE, we provided eBPF support for FUSE. Specifically, we added additional kernel helper functions and designed two new map types to support secure communication between the user-space daemon and kernel extensions, as well as support for passthrough access in read/write. We modified FUSE driver to first invoke registered eBPF handlers (extensions). Passthrough implementation is adopted from WrapFS [54], a wrapper stackable in-kernel file system. Specifically, we modified FUSE driver to pass I/O requests directly to the lower file system.

Since with EXTFUSE developers can install extensions to bypass the user-space daemon and pass I/O requests directly to the lower file system, a malicious process could stack a number of EXTFUSE file systems on top of each other and cause the kernel stack to overflow. To guard against such attacks, we limit the number of EXTFUSE layers that could be stacked on a mount point. We rely on `s_stack_depth` field in the super-block to track the number of stacked layers and check it against `FILESYSTEM_MAX_STACK_DEPTH`, which we limit to two. Table 3 reports the number of lines of code for EXTFUSE. We also modified `libfuse` to allow apps to register kernel extensions.

5 Optimizations

Here, we describe a set of optimizations that can be enabled by leveraging custom kernel extensions in EXTFUSE to implement in-kernel handling of file system requests.

```
1  void handle_lookup(fuse_req_t req, fuse_ino_t pino,
2              const char *name) {
3      /* lookup or create node @cname parent @pino */
4      struct fuse_entry_param e;
5      if (find_or_create_node(req, pino, name, &e)) return;
6  +   lookup_key_t key = {pino, name};
7  +   lookup_val_t val = {0/*not stale*/, &e};
8  +   extfuse_insert_shmap(&key, &val); /* cache this entry */
9      fuse_reply_entry(req, &e);
10 }
```

Figure 2: FUSE daemon `lookup` handler in user space. With EXTFUSE, lines 6-8 (+) enable caching replies in the kernel.

5.1 Customized in-kernel metadata caching

Metadata operations such as `lookup` and `getattr` are frequently issued, and thus form high sources of latency in FUSE file systems [50]. Unlike VFS caches that are only reactive and fixed in functionality, EXTFUSE can be leveraged to proactively cache metadata replies in the kernel. Kernel extensions can be installed to manage and serve subsequent operations from caches without switching to user space.

Example. To illustrate, let us consider the `lookup` operation. It is the most common operation issued internally by the VFS for serving `open()`, `stat()`, and `unlink()` system calls. Each component of the input path string is searched using `lookup` to fetch the corresponding inode data structure. Figure 2 lists code fragment for FUSE daemon handler that serves `lookup` requests in user space (slow path). The FUSE `lookup` API takes two input parameters: the parent node ID and the next path component name. The node ID is a 64-bit integer that uniquely identifies the parent inode. The daemon handler function traverses the parent directory, searching for the child entry corresponding to the next path component. Upon successful search, it populates the `fuse_entry_param` data structure with the node ID and attributes (e.g., size) of the child, and sends it to the FUSE driver, which creates a new inode for the dentry object representing the child entry.

With EXTFUSE, developers could define a `SHashMap` that hosts `fuse_entry_param` replies in the kernel (lines 7-10). A composite key generated from the parent node identifier and the next path component string arguments is used as an index into the map for inserting corresponding replies. Since the map is also accessible to the extensions in the kernel, subsequent requests could be served from the map by installing the EXTFUSE `lookup` extension (fast path). Figure 3 lists its code fragment. The extension uses the same composite key as an index into the hash map to search whether the corresponding `fuse_entry_param` entry exists. If a valid entry is found, the reference count (`nlookup`) is incremented and a reply is sent to the FUSE driver.

Similarly, replies from user space daemon for other metadata operations, such as `getattr`, `getxattr`, and `readlink` could be cached using maps and served in the kernel by respective extensions (Table 4). Network FUSE file systems, such as SshFS [39] and Gluster [40] already perform aggressive metadata caching and batching at client to reduce the number of remote calls to the server. SshFS [39], for example,

```
1  int lookup_extension(extfuse_req_t req, fuse_ino_t pino,
2              const char *name) {
3      /* lookup in map, bail out if not cached or stale */
4      lookup_key_t key = {pino, name};
5      lookup_val_t *val = extfuse_lookup_shmap(&key);
6      if (!val || atomic_read(&val->stale)) return UPCALL;
7      /* EXAMPLE: Android sdcard daemon perm check */
8      if (!check_caller_access(pino, name)) return -EACCES;
9      /* populate output, incr count (used in FUSE_FORGET) */
10     extfuse_reply_entry(req, &val->e);
11     atomic_incr(&val->nlookup, 1);
12     return SUCCESS;
13 }
```

Figure 3: EXTFUSE `lookup` kernel extension that serves valid cached replies, without incurring any context switches. Customized checks could further be included; Android sdcard daemon permission check is shown as an example (see Figure 10).

implements its own directory, attribute, and symlink caches. With EXTFUSE, such caches could be implemented in the kernel for further performance gains.

Invalidation. While caching metadata in the kernel reduces the number of context switches to user space, developers must also carefully invalidate replies, as necessary. For example, when a file (or dir) is deleted or renamed, the corresponding cached `lookup` replies must be invalidated. Invalidations can be performed in user space by the relevant request handlers or in the kernel by installing their extensions before new changes are made. However, the former case may introduce race conditions and produce incorrect results because all requests to user space daemon are queued up by the FUSE driver, whereas requests to the extensions are not. Cached `lookup` replies can be invalidated in extensions for `unlink`, `rmdir`, and `rename` operations. Similarly, when attributes or permissions on a file change, cached `getattr` replies can be invalidated in `setattr` extension. Our design ensures race-free invalidation by executing the extensions before forwarding requests to user space daemon where the changes may be made.

Advantages over VFS caching. As previously mentioned, recent optimizations added to FUSE framework leverage VFS caches to reduce user-kernel context switching. For instance, by specifying non-zero `entry_valid` and `attr_valid` timeout values, dentries and inodes cached by the VFS from previous `lookup` operations could be utilized to serve subsequent `lookup` and `getattr` requests, respectively. However, the VFS offers no control to the user file system over the cached data. For example, if the file system is mounted without the `default_permissions` parameter, VFS caching of inodes introduces a security bug [21]. This is because the cached permissions are only checked for first accessing user. In contrast, with EXTFUSE, developers can define their own metadata caches and install custom code to manage them. For instance, extensions can perform `uid`-based access permission checks before serving requests from the caches to obviate the aforementioned security issue (Figure 10).

Additionally, unlike VFS caches that are only reactive, EXTFUSE enables proactive caching. For example, since a `readdir` request is expected after an `opendir` call, the user-space daemon could proactively cache directory entries in the

Metadata	Map Key	Map Value	Caching Operations	Serving Extensions	Invalidation Operations
Inode	<nodeID, name>	fuse_entry_param	`lookup, create, mkdir, mknod`	*`lookup`*	`unlink, rmdir, rename`
Attrs	<nodeID>	fuse_attr_out	`getattr, lookup`	*`getattr`*	`setattr, unlink, rmdir`
Symlink	<nodeID>	link path	`symlink, readlink`	*`readlink`*	`unlink`
Dentry	<nodeID>	fuse_dirent	`opendir, readdir`	*`readdir`*	`releasedir, unlink, rmdir, rename`
XAttrs	<nodeID, label>	xattr value	`open, getxattr, listxattr`	*`getattr`*, *`listxattr`*	`close, setxattr, removexattr`

Table 4: Metadata can be cached in the kernel using eBPF maps by the user-space daemon and served by kernel extensions.

kernel by inserting them in a BPF map while serving `opendir` requests to reduce transitions to user space. Alternatively, similar to read-ahead optimization, proactive caching of subsequent directory entries could be performed during the first `readdir` call to the user-space daemon. Memory occupied by cached entries could then be freed by the `releasedir` handler in user space that deletes them from the map. Similarly, security labels on a file could be cached during the `open` call to user space and served in the kernel by `getxattr` extensions. Nonetheless, since eBPF maps consume kernel memory, developers must carefully manage caches and limit the number of map entries to keep memory usage under check.

5.2 Passthrough I/O for stacking functionality

Many user file systems are stackable with a thin layer of functionality that does not require complex processing in the user-space. For example, LoggedFS [16] filters requests that must be logged, logs them as needed, and then simply forwards them to the lower file system. User-space union file systems, such as MergerFS [20] determine the backend host file in `open` and redirects I/O requests to it. BindFS [35] mirrors another mount point with custom permissions checks. Android sdcard daemon performs access permission checks and emulates the case-insensitive behavior of FAT only in metadata operations (e.g., `lookup`, `open`, etc.), but forwards data I/O requests directly to the lower file system. For such simple cases, the FUSE API proves to be too low-level and incurs unnecessarily high overhead due to context switching.

With EXTFUSE, `read/write` I/O requests can take the fast path and directly be forwarded to the lower (host) file system without incurring any context-switching if the complex slow-path user-space logic is not needed. Figure 4 shows how the user-space daemon can install the lower file descriptor in `InodeMap` while handling `open()` system call for notifying the EXTFUSE driver to store a reference to the lower inode kernel object. With the `custom_filtering_logic(path)` condition, this can be done selectively; for example, if access permission checks pass in Android sdcard daemon. Similarly, BindFS and MergerFS can adopt EXTFUSE to avail passthrough optimization. The `read/write` kernel extensions can check in `InodeMap` to detect whether the target file is setup for passthrough access. If found, EXTFUSE driver can be instructed with a special return code to directly forward the I/O request to the lower file system with the corresponding lower inode object as parameter. Figure 5 shows a template `read` extension. Kernel extensions can include additional logic or checks before returning. For instance, LoggedFS `read/write` extensions can filter and log operations, as needed §6.2.

```
1  void handle_open(fuse_req_t req, fuse_ino_t ino,
2          const struct fuse_open_in *in) {
3      /* file represented by @ino inode num */
4      struct fuse_open_out out; char path[PATH_MAX];
5      int len, fd = open_file(ino, in->flags, path, &out);
6      if (fd > 0 && custom_filtering_logic(path)) {
7  +       /* install fd in inode map for pasthru */
8  +       imap_key_t key = out->fh;
9  +       imap_val_t val = fd; /* lower fd */
10 +       extfuse_insert_imap(&key, &val);
11 }   }
```

Figure 4: FUSE daemon `open` handler in user space. With EXTFUSE, lines 7-9 (+) enable passthrough access on the file.

```
1  int read_extension(extfuse_req_t req, fuse_ino_t ino,
2          const struct fuse_read_in *in) {
3      /* lookup in inode map, passthrough if exists */
4      imap_key_t key = in->fh;
5      if (!extfuse_lookup_imap(&key)) return UPCALL;
6      /* EXAMPLE: LoggedFS log operation */
7      log_op(req, ino, FUSE_READ, in, sizeof(*in));
8      return PASSTHRU; /* forward req to lower FS */
9  }
```

Figure 5: The EXTFUSE `read` kernel extension returns `PASSTHRU` to forward request directly to the lower file system. Custom thin functionality could further be pushed in the kernel; LoggedFS logging function is shown as an example (see Figure 11).

6 Evaluation

To evaluate EXTFUSE, we answer the following questions:

- **Baseline Performance.** How does an EXTFUSE implementation of a file system perform when compared to its in-kernel and FUSE implementations? (§6.1)
- **Use cases.** What kind of existing FUSE file systems can benefit from EXTFUSE and what performance improvements can they expect? (§6.2)

6.1 Performance

To measure the baseline performance of EXTFUSE, we adopted the simple no-ops (null) stackable FUSE file system called Stackfs [50]. This user-space daemon serves all requests by forwarding them to the host (lower) file system. It includes all recent FUSE optimizations (Table 5). We evaluate Stackfs under all possible EXTFUSE configs listed in Table 5. Each config represents a particular level of performance that could potentially be achieved, for example, by caching metadata in the kernel or directly passing `read/write` requests through the host file system for stacking functionality. To put our results in context, we compare our results with EXT4 and

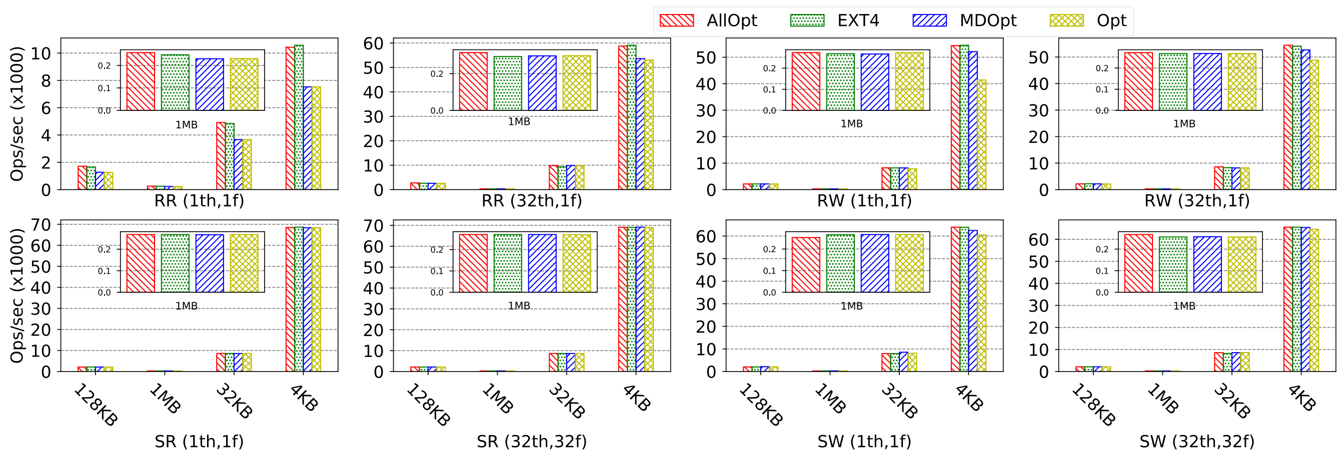


Figure 6: Throughput(ops/sec) for EXT4 and FUSE/ExtFUSE Stackfs (w/ xattr) file systems under different configs (Table 5) as measured by Random Read(RR)/Write(RW), Sequential Read(SR)/Write(SW) Filebench [48] data micro-workloads with IO Sizes between 4KB-1MB and settings *N*th: *N* threads, *N*f: *N* files. We use the same workloads as in [50].

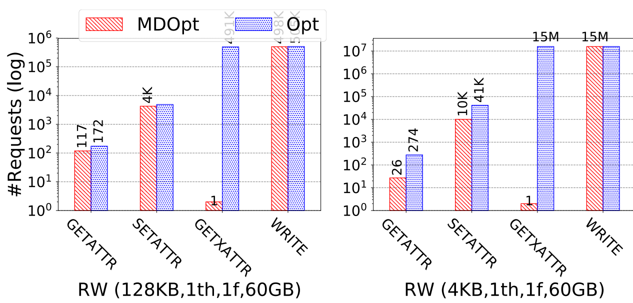


Figure 7: Number of file system request received by the daemon in FUSE/ExtFUSE Stackfs (w/ xattr) under workloads in Figure 6. Only a few relevant request types are shown.

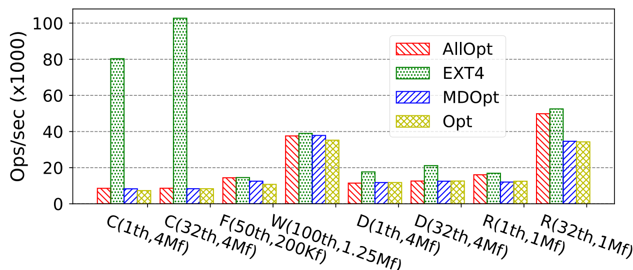


Figure 8: Throughput(Ops/sec) for EXT4 and FUSE/ExtFUSE Stackfs (w/ xattr) under different configs (Table 5) as measured by Filebench [48] Creation(C), Deletion(D), Reading(R) metadata micro-workloads on 4KB files and FileServer(F), WebServer(W) macro-workloads with settings *N*th:*N* threads, *N*f:*N* files.

the optimized FUSE implementation of Stackfs (`Opt`).

Testbed. We use the same experiments and settings as in [50]. Specifically, we used EXT4 because of its popularity as the host file system and ran benchmarks to evaluate. However, since FUSE performance problems were reported to be more prominent with a faster storage medium, we only carry out our experiments with SSDs. We used a Samsung 850 EVO 250GB SSD installed on an Asus machine with Intel Quad-Core i5-3550 3.3 GHz and 16GB RAM, running Ubuntu 16.04.3. Further, to minimize any variability, we formatted the SSD before each experiment and disabled EXT4 lazy inode initialization. To evaluate file systems that implement `xattr` operations for handling security labels (e.g., in Android), our implementation of `Opt` supports xattrs, and thus differs from the implementation in [50].

Config	File System	Optimizations
`Opt` [50]	FUSE	128K Writes, Splice, WBCache, MltThrd
`MDOpt`	ExtFUSE	`Opt` + Caches `lookup`, `attrs`, `xattrs`
`AllOpt`	ExtFUSE	`MDOpt` + Pass `R/W` reqs through host FS

Table 5: Different Stackfs configs evaluated.

Workloads. Our workload consists of Filebench [48] micro and synthetic macro benchmarks to test each config with metadata- and data-heavy operations across a wide range of I/O sizes and parallelism settings. We measure the low-level throughput (ops/sec). Our macro-benchmarks consist of a synthetic file server and web server.

Micro Results. Figure 6 shows the results of micro workload under different configs listed in Table 5.

Reads. Due to the default 128KB read-ahead feature of FUSE, the sequential read throughput on a single file with a single thread for all I/O sizes and under all Stackfs configs remained the same. Multi-threading improved for the sequential read benchmark with 32 threads and 32 files. Only one request was generated per thread for `lookup` and `getattr` operations. Hence, metadata caching in `MDOpt` was not effective. Since FUSE `Opt` performance is already at par with EXT4, the passthrough feature in `AllOpt` was not utilized.

Unlike sequential reads, small random reads could not take advantage of the read-ahead feature of FUSE. Additionally, 4KB reads are not spliced and incur data copying across user-kernel boundary. With 32 threads operating on a single file, the throughput improves due to multi-threading in `Opt`. However, degradation is observed with 4KB reads. `AllOpt` passes all reads through EXT4, and hence offers near-native throughput. In some cases, the performance was slightly better than

EXT4. We believe that this minor improvement is due to double caching at the VFS layer. Due to a single request per thread for metadata operations, no improvement was seen with EXTFUSE metadata caching.

Writes. During sequential writes, the 128K big writes and writeback caching in `Opt` allow the FUSE driver to batch small writes (up to 128KB) together in the page cache to offer a higher throughput. However, random writes are not batched. As a result, more `write` requests are delivered to user space, which negatively affects the throughput. Multiple threads on a single file perform better for requests bigger than 4KB as they are spliced. With EXTFUSE `AllOpt`, all writes are passed through the EXT4 file system to offer improved performance.

Write throughput degrades severely for FUSE file systems that support extended attributes because the VFS issues a `getxattr` request before every `write`. Small I/O requests perform worse as they require more `write`, which generate more `getxattr` requests. `Opt` random writes generated 30x fewer `getxattr` requests for 128KB compared to 4KB writes, resulting in a 23% decrease in the throughput of 4KB writes.

In contrast, `MDOpt` caches the `getxattr` reply in the kernel upon the first call, and serves subsequent `getxattr` requests without incurring further transitions to user space. Figure 7 compares the number of requests received by the user-space daemon in `Opt` and `MDOpt`. Caching replies reduced the overhead for 4KB workload to less than 5%. Similar behavior was observed with both sequential writes and random writes.

Macro Results. Figure 8 shows the results of macro-workloads and synthetic server workloads emulated using Filebench under various configs. Neither of the EXTFUSE configs offer improvements over FUSE `Opt` under creation and deletion workloads as these metadata-heavy workloads created and deleted a number of files, respectively. This is because no metadata caching could be utilized by `MDOpt`. Similarly, no passthrough writes were utilized with `AllOpt` since 4KB files were created and closed in user space. In contrast, the File and Web server workloads under EXTFUSE utilized both metadata caching and passthrough access features and improved performance. We saw a 47%, 89%, and 100% drop in `lookup`, `getattr`, and `getattr` requests to user space under `MDOpt`, respectively, when configured to cache up to 64K for each type of request. `AllOpt` further enabled passthrough `read/write` requests to offer near native throughput for both macro reads and server workloads.

Real Workload We also evaluated EXTFUSE with two real workloads, namely kernel decompression and compilation of 4.18.0 Linux kernel. We created three separate caches for hosting `lookup`, `getattr`, and `getxattr` replies. Each cache could host up to 64K entries, resulting in allocation of up to a total of 50MB memory when fully populated.

The kernel compilation `make tinyconfig; make -j4` experiment on our test machine (see §6) reported a 5.2% drop in compilation time, from 39.74 secs under FUSE `Opt` to 37.68 secs with EXTFUSE `MDOpt`, compared to 30.91 secs with EXT4. This was due to over 75%, 99%, and 100% decrease in `lookup`, `getattr`, and `getxattr` requests to user space, respectively (Figure 9). `getxattr` replies were proactively cached while handling `open` requests; thus, no transitions to user space were observed for serving xattr requests. With EXTFUSE `AllOpt`, the compilation time further dropped to 33.64 secs because of 100% reduction in `read` and `write` requests to user space.

In contrast, the kernel decompression `tar xf` experiment reported a 6.35% drop in the completion time, from 11.02 secs under FUSE `Opt` to 10.32 secs with EXTFUSE `MDOpt`, compared to 5.27 secs with EXT4. With EXTFUSE `AllOpt`, the decompression time further dropped to 8.67 secs due to 100% reduction in `read` and `write` requests to user space, as shown in Figure 9. Nevertheless, reducing the number of cached entries for metadata requests to 4K resulted in a decompression time of 10.87 secs (25.3% increase) due to 3,555 more `getattr` requests to user space. This suggests that developers must efficiently manage caches.

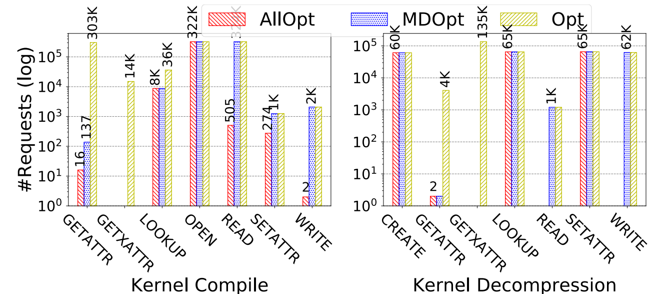


Figure 9: Linux kernel 4.18.0 untar (decompress) and compilation time taken with StackFS under FUSE and EXTFUSE settings. Number of metadata and I/O requests are reduced with EXTFUSE.

6.2 Use cases

We ported four real-world stackable FUSE file systems, namely LoggedFS, Android sdcard daemon, MergerFS, and BindFS to EXTFUSE and enabled both metadata caching §5.1 and passthrough I/O §5.2 optimizations.

File System	Functionality	Ext Loc
StackFS [50]	No-ops File System	664
BindFS [35]	Mirroring File System	792
Android sdcard [24]	Perm checks & FAT Emu	928
MergerFS [20]	Union File System	686
LoggedFS [16]	Logging File System	748

Table 6: Lines of code (Loc) of kernel extensions required to adopt EXTFUSE for existing FUSE file systems. We added support for metadata caching as well as R/W passthrough.

As EXTFUSE allows file systems to retain their existing FUSE daemon code as the default slow path, adopting EXTFUSE for real-world file systems is easy. On average, we made less than 100 lines of changes to the existing FUSE code to invoke EXTFUSE helper library functions for manipulating kernel extensions, including maps. We added ker-

nel extensions to support metadata caching as well as I/O passthrough. Overall, it required fewer than 1000 lines of new code in the kernel Table 6. We now present detailed evaluation of Android sdcard daemon and LoggedFS to present an idea on expected performance improvements.

Android sdcard daemon. Starting version 3.0, Android introduced the support for FUSE to allow a large part of internal storage (e.g., `/data/media`) to be mounted as external FUSE-managed storage (called `/sdcard`). Being large in size, `/sdcard` hosts user data, such as videos and photos as well as any auxiliary Opaque Binary Blobs (OBB) needed by Android apps. The FUSE daemon enforces permission checks in metadata operations (e.g.,`lookup`, etc.) on files under `/sdcard` to enable multi-user support and emulates case-insensitive FAT functionality on the host (e.g., EXT4) file system. OBB files are compressed archives and typically used by games to host multiple small binaries (e.g. shade rs, textures) and multimedia objects (e.g. images, etc.).

However, FUSE incurs high runtime performance overhead. For instance, accessing OBB archive content through the FUSE layer leads to high launch latency and CPU utilization for gaming apps. Therefore, Android version 7.0 replaced sdcard daemon with with an in-kernel file system called `SDCardFS` [24] to manage external storage. It is a wrapper (thin) stackable file system based on WrapFS [54] that enforces permission checks and performs FAT emulation in the kernel. As such, it imposes little to no overhead compared to its user-space implementation. Nevertheless, it introduces security risks and maintenance costs [12].

We ported Android sdcard FUSE daemon to EXTFUSE framework. First, we leverage eBPF kernel helper functions to push metadata checks into the kernel. For example, we embed access permission check (Figure 10) in `lookup` kernel extension to validate access before serving `lookup` replies from the cache (Figure 3). Similar permission checks are performed in the kernel to validate accesses to files under `/sdcard` before serving cached `getattr` requests . We also enabled passthrough on `read/write` using `InodeMap`.

We evaluated its performance on a 1GB RAM HiKey620 board [1] with two popular game apps containing OBB files of different sizes. Our results show that under `AllOpt` passthrough mode the app launch latency and the corresponding peak CPU consumption reduces by over 90% and 80%, respectively. Furthermore, we found that the larger the OBB file, the more penalty is incurred by FUSE due to many more small files in the OBB archive.

LoggedFS is a FUSE-based stackable user-space file system. It logs every file system operation for monitoring purposes. By default it writes to syslog buffer and logs all operations (e.g., open, read, etc.). However, it can be configured to write to a file or log selectively. Despite being a simple file system, it has a very important use case. Unlike existing monitoring mechanisms (e.g., Inotify [31]) that suffer from a host of limitations [10], LoggedFS can reliably post all file system events. Various apps, such as file system indexers, backup tools, Cloud storage clients such as Dropbox, integrity checkers, and antivirus software subscribe to file system events for efficiently tracking modifications to files.

App Stats		CPU (%)		Latency (ms)	
Name	**OBB Size**	**D**	**P**	**D**	**P**
Disney Palace Pets 5.1	374MB	20	2.9	2235	1766
Dead Effect 4	1.1GB	20.5	3.2	8895	4579

Table 7: App launch latency and peak CPU consumption of `sdcard` daemon under default (D), and passthrough (P) settings on Android for two popular games. In passthrough mode, the FUSE driver never forwards `read/write` requests to user space, but always passes them through the host (EXT4) file system. See Table 5 for config details.

```
1  bool check_caller_access_to_name(int64_t key, const char *name) {
2      /* define a shmap for hosting permissions */
3      int *val = extfuse_lookup_shmap(&key);
4      /* Always block security-sensitive files at root */
5      if (!val || *val == PERM_ROOT) return false;
6      /* special reserved files */
7      if (!strncasecmp(name, "autorun.inf", 11) ||
8          !strncasecmp(name, ".android_secure", 15) ||
9          !strncasecmp(name, "android_secure", 14))
10         return false;
11     return true;
12 }
```

Figure 10: Android `sdcard` permission checks EXTFUSE code.

We ported LoggedFS to EXTFUSE framework. Figure 11 shows the common logging code that is called from various extensions, which serve requests in the kernel (e.g., `read` extension Figure 5). To evaluate its performance, we ran the FileServer macro benchmark with synthetic a workload of 200,000 files and 50 threads from Filebench suite. We found over 9% improvement in throughput under `MDOpt` compared to FUSE `Opt` due to 53%, 99%, and 100% fewer `lookup`, `getattr`, and `getxattr` requests to user space, respectively. Figure 12 shows the results. `AllOpt` reported an additional 20% improvement by directly forwarding all `read/write` requests to the host file system, offering near-native throughput.

```
1  void log_op(extfuse_req_t req, fuse_ino_t ino,
2          int op, const void *arg, int arglen) {
3      struct data { /* log record */
4          u32 op; u32 pid; u64 ts; u64 ino; char data[MAXLEN];};
5      /* example filter: only whitelisted UIDs in map */
6      u16 uid = bpf_get_current_uid_gid();
7      if (!extfuse_lookup_shmap(uid_wlist, &uid)) return;
8      /* log opcode, timestamp(ns) and requesting process */
9      data.opcode = op; data.ts  = bpf_ktime_get_ns();
10     data.pid = bpf_get_current_pid_tgid(); data.ino = ino;
11     memcpy(data.data, arg, arglen);
12     /* submit to per-cpu mmap'd ring buffer */
13     u32 key = bpf_get_smp_processor_id();
14     bpf_perf_event_output(req, &buf, &key, &data, sizeof(data));
15 }
```

Figure 11: LoggedFS kernel extension that logs requests.

7 Discussion

Future use cases. Given negligible overhead of EXTFUSE and direct passthrough access to the host file system for stacking incremental functionality, multiple app-defined "thin" file system functions (e.g., security checks, logging, etc.) can be

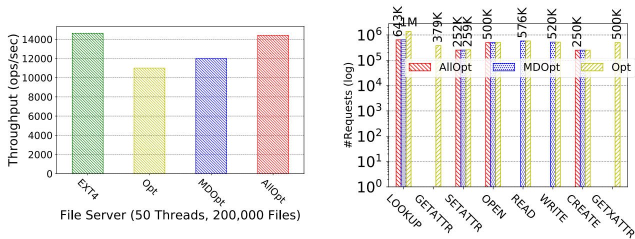


Figure 12: LoggedFS performance measured by Filebench FileServer benchmark under EXT4, FUSE, and EXTFUSE. Fewer metadata and I/O requests were delivered to user space with EXTFUSE.

stacked with low overhead, which otherwise would have been very expensive in user space with FUSE.

Safety. EXTFUSE sandboxes untrusted user extensions to guarantee safety. For example, the eBPF runtime allows access to only a few, simple non-blocking kernel helper functions. Map data structures are of fixed size. Extensions are not allowed to allocate memory or directly perform any I/O operations. Even so, EXTFUSE offers significant performance boost across a number of use cases §6.2 by offloading simple logic in the kernel. Nevertheless, with EXTFUSE, user file systems can retain their existing slow-path logic for performing complex operations, such as encryption in user space. Future work can extend the EXTFUSE framework to take advantage of existing generic in-kernel services such as VFS encryption and compression APIs to even serve requests that require such complex operations entirely in the kernel.

8 Related Work

Here, we compare our work with related existing research.

User File System Frameworks. There exists a number of frameworks to develop user file systems. A number of user file systems have been implemented using NFS loopback servers [19]. UserFS [14] exports generic VFS-like file system requests to the user space through a file descriptor. Arla [53] is an AFS client system that lets apps implement a file system by sending messages through a device file interface `/dev/xfs0`. Coda file system [42] exported a similar interface through `/dev/cfs0`. NetBSD provides Pass-to-Userspace Framework FileSystem (PUFFS). Mazières et al. proposed a C++ toolkit that exposes a NFS-like interface for allowing file systems to be implemented in user space [33]. UFO [3] is a global file system implemented in user space by introducing a specialized layer between the apps and the OS that intercepts file system calls.

Extensible Systems. Past works have explored the idea of letting apps extend system services at runtime to meet their performance and functionality needs. SPIN [6] and VINO [43] allow apps to safely insert kernel extensions. SPIN uses a type-safe language runtime, whereas VINO uses software fault isolation to provide safety. ExoKernel [13] is another OS design that lets apps define their functionality. Systems such as ASHs [17, 51] and Plexus [15] introduced the concept of network stack extension handlers inserted into the kernel. SLIC [18] extends services in monolithic OS using interposition to enable incremental functionality and composition. SLIC assumes that extensions are trusted. EXTFUSE is a framework that allows user file systems to add "thin" extensions in the kernel that serve as specialized interposition layers to support both in-kernel and user space processing to co-exist in monolithic OSes.

eBPF. EXTFUSE is not the first system to use eBPF for safe extensibility. eXpress DataPath (XDP) [27] allows apps to insert eBPF hooks in the kernel for faster packet processing and filtering. Amit et al. proposed Hyperupcalls [4] as eBPF helper functions for guest VMs that are executed by the hypervisor. More recently, SandFS [7] uses eBPF to provide an extensible file system sandboxing framework. Like EXTFUSE, it also allows unprivileged apps to insert custom security checks into the kernel.

FUSE. File System Translator (FiST) [55] is a tool for simplifying the development of stackable file system. It provides *boilerplate* template code and allows developers to only implement the core functionality of the file system. FiST does not offer safety and reliability as offered by user space file system implementation. Additionally, it requires learning a slightly simplified file system language that describes the operation of the stackable file system. Furthermore, it only applies to stackable file systems.

Narayan et al. [34] combined in-kernel stackable FiST driver with FUSE to offload data from I/O requests to user space to apply complex functionality logic and pass processed results to the lower file system. Their approach is only applicable to stackable file systems. They further rely on static per-file policies based on extended attributes labels to enable or disable certain functionality. In contrast, EXTFUSE downloads and safely executes thin extensions from user file systems in the kernel that encapsulate their rich and specialized logic to serve requests in the kernel and skip unnecessary user-kernel switching.

9 Conclusion

We propose the idea of extensible user file systems and present the design and architecture of EXTFUSE, an extension framework for FUSE file system. EXTFUSE allows FUSE file systems to define "thin" extensions along with auxiliary data structures for specialized handling of low-level requests in the kernel while retaining their existing complex logic in user space. EXTFUSE provides familiar FUSE-like APIs and abstractions for easy adoption. We demonstrate its practical usefulness, suitability for adoption, and performance benefits by porting and evaluating existing FUSE implementations.

10 Acknowledgments

We thank our shepherd, Dan Williams, and all anonymous reviewers for their feedback, which improved the content of this paper. This work was funded in part by NSF CPS program Award #1446801, and a gift from Microsoft Corp.

References

[1] 96boards. Hikey (lemaker) development boards, May 2019.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[3] Albert D Alexandrov, Maximilian Ibel, Klaus E Schauser, and Chris J Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX Annual Technical Conference (ATC)*, pages 6–6, Anaheim, California, January 1997.

[4] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 97–112, Boston, MA, July 2018.

[5] Starr Andersen and Vincent Abella. Changes to Functionality in Windows XP Service Pack 2, Part 3: Memory Protection Technologies, 2004. https://technet.microsoft.com/en-us/library/bb457155.aspx.

[6] 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 *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[7] Ashish Bijlani and Umakishore Ramachandran. A lightweight and fine-grained file system sandboxing framework. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, Jeju Island, South Korea, August 2018.

[8] Ceph. Ceph kernel client, April 2018. https://github.com/ceph/ceph-client.

[9] Open ZFS Community. ZFS on Linux. https://zfsonlinux.org, April 2018.

[10] J. Corbet. Superblock watch for fsnotify, April 2017.

[11] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, pages 27–27, Boston, MA, June–July 2004.

[12] Exploit Database. Android - sdcardfs changes current->fs without proper locking, 2019.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[14] Jeremy Fitzhardinge. Userfs, March 2018. http://www.goop.org/~jeremy/userfs/.

[15] Marc E. Fiuczynski and Briyan N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Diego, CA, January 1996.

[16] R. Flament. LoggedFS - Filesystem monitoring with Fuse, March 2018. https://rflament.github.io/loggedfs/.

[17] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, 2002.

[18] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. Slic: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, New Orleans, Louisiana, June 1998.

[19] David K Gifford, Pierre Jouvelot, Mark A Sheldon, et al. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–25, Pacific Grove, CA, October 1991.

[20] A Featureful Union Filesystem, March 2018. https://github.com/trapexit/mergerfs.

[21] LibFuse | GitHub. Without 'default_permissions', cached permissions are only checked on first access, 2018. https://github.com/libfuse/libfuse/issues/15.

[22] Gluster. libgfapi, April 2018. http://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/libgfapi/.

[23] Gnu hurd, April 2018. www.gnu.org/software/hurd/hurd.html.

[24] Storage | Android Open Source Project, September 2018. https://source.android.com/devices/storage/.

[25] John H Hartman and John K Ousterhout. Performance measurements of a multiprocessor sprite kernel. In *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, pages 279–288, Anaheim, CA, 1990.

[26] eBPF: extended Berkley Packet Filter, 2017. https://www.iovisor.org/technology/ebpf.

[27] IOVisor. Xdp - io visor project, May 2019.

[28] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for fuse-based distributed storage. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 760–765, 2012.

[29] Antti Kantee. puffs-pass-to-userspace framework file system. In *Proceedings of the Asian BSD Conference (AsiaBSDCon)*, Tokyo, Japan, March 2007.

[30] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[31] Robert Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005:8, 2005.

[32] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[33] David Mazières. A Toolkit for User-Level File Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, pages 261–274, June 2001.

[34] S. Narayan, R. K. Mehta, and J. A. Chandy. User space storage system stack modules with file level control. In *Proceedings of the Linux Symposium*, pages 189–196, Ottawa, Canada, July 2010.

[35] Martin Pärtel. bindfs, 2018. https://bindfs.org.

[36] PaX Team. PaX address space layout randomization (ASLR), 2003. https://pax.grsecurity.net/docs/aslr.txt.

[37] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all. In *Proceedings of the 10th ACM International on Systems and Storage Conference*, pages 9:1–9:12, Haifa, Israel, May 2017.

[38] Nikolaus Rath. List of fuse file systems, 2011. https://github.com/libfuse/libfuse/wiki/Filesystems.

[39] Nicholas Rauth. A network filesystem client to connect to SSH servers, April 2018. https://github.com/libfuse/sshfs.

[40] Gluster, April 2018. http://gluster.org.

[41] Kai Ren and Garth Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 145–156, San Jose, CA, June 2013.

[42] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[43] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. An introduction to the architecture of the vino kernel. Technical report, Technical Report 34-94, Harvard Computer Center for Research in Computing Technology, October 1994.

[44] Helgi Sigurbjarnarson, Petur O. Ragnarsson, Juncheng Yang, Ymir Vigfusson, and Mahesh Balakrishnan. Enabling space elasticity in storage systems. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 6:1–6:11, Haifa, Israel, June 2016.

[45] David C Steere, James J Kistler, and Mahadev Satyanarayanan. Efficient user-level file cache management on the sun vnode interface. In *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, 1990.

[46] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Refuse to Crash with Re-FUSE. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, Salzburg, Austria, April 2011.

[47] M. Szeredi and N.Rauth. Fuse - filesystems in userspace, 2018. https://github.com/libfuse/libfuse.

[48] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[49] Ungureanu, Cristian and Atkin, Benjamin and Aranya, Akshat and Gokhale, Salil and Rago, Stephen and Całkowski, Grzegorz and Dubnicki, Cezary and Bohra, Aniruddha. HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System. In *10th USENIX Conference on File and Storage Technologies (FAST) (FAST 10)*, pages 17–17, San Jose, California, February 2010.

[50] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*, Santa Clara, CA, February 2017.

[51] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. In *Proceedings of the 7th ACM SIGCOMM*, Palo Alto, CA, August 1996.

[52] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, Seattle, WA, November 2006.

[53] Assar Westerlund and Johan Danielsson. Arla: a free afs client. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, pages 32–32, New Orleans, Louisiana, June 1998.

[54] E. Zadok, I. Bădulescu, and A. Shender. Extending File Systems Using Stackable Templates". In *Proceedings of the 1999 USENIX Annual Technical Conference (ATC)*, pages 57–70, June 1999.

[55] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (ATC)*, June 2000.

[56] Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traeger. Efficient and safe execution of user-level code in the kernel. In *Parallel and Distributed Processing Symposium 19th IEEE International*, pages 8–8, 2005.

[57] ZFS-FUSE, April 2018. https://github.com/zfs-fuse/zfs-fuse.

FlexGroup Volumes: A Distributed WAFL File System

Ram Kesavan,[*] Jason Hennessey, Richard Jernigan, Peter Macko,

Keith A. Smith, Daniel Tennant, and Bharadwaj V. R., *NetApp, Inc.*

Abstract

The rapid growth of customer applications and datasets has led to demand for storage that can scale with the needs of modern workloads. We have developed FlexGroup volumes to meet this need. FlexGroups combine local WAFL® file systems in a distributed storage cluster to provide a single namespace that seamlessly scales across the aggregate resources of the cluster (CPU, storage, etc.) while preserving the features and robustness of the WAFL file system.

In this paper we present the FlexGroup design, which includes a new *remote access layer* that supports distributed transactions and the novel heuristics used to balance load and capacity across a storage cluster. We evaluate FlexGroup performance and efficacy through lab tests and field data from over 1,000 customer FlexGroups.

1 Introduction

With each new generation of hardware, computers become faster and more powerful. CPUs have more cores, memory is cheaper, networks are faster, and storage devices hold more data. Despite these advances, however, many modern applications require far more resources than a single machine can provide, leading to an explosion in the use of distributed applications and systems.

Network-attached storage solutions have evolved similarly from single-node to distributed systems. NetApp® Write Anywhere File Layout (WAFL) [11] was launched more than 20 years ago as a single-node, single-volume file system. Over time, we increased WAFL flexibility and scale by allowing many file systems per node [7], and by scaling performance with increasing core counts [5]. Today we have hundreds of thousands of storage controllers at customers' sites. But like other file systems, our single-node scale and performance have been limited by the resources (storage, memory, CPU, network) of a single machine.

This paper describes *FlexGroups*, a new feature that *automatically* balances capacity and load across the nodes of our storage cluster. A FlexGroup combines volumes from multiple nodes of a cluster into a single client-visible namespace and allows any directory entry to point to any inode on any of the member volumes. The FlexGroup selects a location for each new file or directory using heuristics that attempt to balance capacity and load. The result is a single file system that scales across the resources of an entire storage cluster.

Within a storage cluster, FlexGroups use a new *Remote Access Layer* (RAL) to perform transactional updates across multiple member volumes (for example, a directory entry on one node and the target inode on a different node). Because operations that span multiple FlexVols® necessarily introduce overhead, the FlexGroup heuristics dynamically adjust the levels of remote inode placement to minimize that overhead while still achieving a balanced system.

FlexGroups impose performance overhead on metadata operations that create or traverse cross-node links in the namespace. But the more costly of these operations, such as `MKDIR`, are relatively rare, and the overheads are modest compared to the larger latencies of I/O to SSDs or HDDs. On large mixed workloads, a FlexGroup performs comparably to a similarly sized group of individual volumes while providing the benefits of automatic scaling and balancing of the data and load across nodes.

This paper makes several contributions. We present a scalable distributed file system architecture that builds on an underlying single-node file system while preserving all of its features. We describe low-cost heuristics that dynamically balance load and capacity with little performance overhead. Finally, we analyze more than 1,000 customer FlexGroup deployments to evaluate our load balancing heuristics, understand challenging use cases, and identify improvements to FlexGroup heuristics.

2 The Building Blocks

A FlexGroup is a distributed file system built from a cluster of nodes running our storage operating system, NetApp Data ONTAP®, which includes WAFL [11], our proprietary copy-on-write (COW) file system. A FlexGroup is com-

[*]Ram Kesavan is currently at Google.

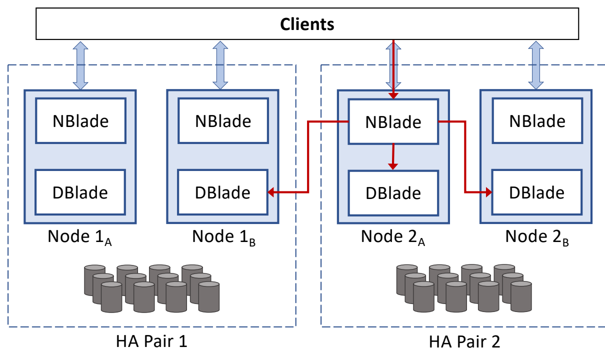


Figure 1: Arrows indicate some possible routing paths for operations. The NBlade in node 2_A may route a request to the DBlade on the same node, on the HA partner node (2_B), or to another node in the cluster (1_B).

posed of multiple single-node file systems called FlexVols; a FlexVol [7] is an exportable WAFL file system. As in other COW file systems [24, 30], every modified block of data or metadata in WAFL is written to a new location; only the superblock is ever written in place. The file system supports many enterprise features, such as snapshots [18, 19], replication [29], storage efficiency (compression, deduplication, etc.), and encryption.

A cluster of ONTAP [25] nodes is organized as multiple high-availability (HA) pairs of nodes. Each pool of storage is accessible by a different HA pair. A node is composed primarily of two software modules [8]: an *NBlade* and a *DBlade*. The NBlade is composed of the networking and protocol stacks (NFS, SMB, iSCSI, NVMe, etc.) that communicate with clients. The DBlade is composed of the WAFL file system, RAID [2, 9] module, and storage drivers to interact with the storage media. The storage pool dedicated to an HA pair is partitioned into *aggregates*, each of which can house hundreds of FlexVols. The storage devices in an aggregate are collected into RAID groups to protect against device failures [28]. Each aggregate is managed in an active-passive manner by the nodes of its HA pair. Thus the DBlade on each node serves requests for one or more aggregates while also acting as a standby to take over the HA partner's aggregates if the partner fails.

ONTAP exports file system namespaces over virtual interfaces (VIFs) that are mapped to nodes in the cluster. A client accesses the namespace within a FlexVol over NFS or SMB by connecting to a node via the corresponding VIF [8]. As illustrated in Fig. 1, each request is routed from the receiving node's NBlade to the DBlade of the node owning the FlexVol that contains the requested data. The NBlades consult a consistent, replicated database of volume information, which maps volume IDs to DBlades. Responses are routed back from a DBlade to the client via the NBlade that is connected to the client. Any operation that is routed to a different node pays a small latency penalty compared to one that is routed to the same node [8].

WAFL accelerates performance by placing a write-ahead log in NVRAM [11, 33] and provides a fast failover capability by mirroring this log to its HA partner's NVRAM. If a node (1_A) fails, the HA partner (1_B) takes ownership of its network interfaces and storage, replays its NVRAM log (aka NVLog), and starts servicing operations to those FlexVols almost instantly, thereby minimizing client outage. Even in the rare event that NVLog is lost—for example, if one node "panics" and the NVRAM devices on both nodes of the HA-pair are damaged—the COW nature of updates guarantees the consistency of the persistent file systems on the failed node [11, 24, 30]; only state from operations logged in the last few seconds is lost. In other words, no (fsck-style) repair of the file system is needed in such cases.

3 FlexGroup Volumes

ONTAP with FlexVols had been commercially successful with a wide customer deployment for several years before we tackled the problem of building a distributed namespace that seamlessly scales across the storage pools and compute resources of all nodes in a cluster. ONTAP already allowed the administrator to manually *junction* together multiple FlexVols in a cluster to export a namespace that spanned multiple nodes, but it was impossible to build an efficient junctioned namespace because avoiding conditions of imbalance—storage capacity or load—required a priori knowledge of future behavior of the applications. Customer deployments need directory namespaces to autonomously consume capacity and compute resources from nodes in the cluster with minimal administrator involvement.

Customers have high requirements of an enterprise-grade distributed file system: resiliency, availability, ease of administration, and other "standard" features, such as snapshots, replication, cloning, compression, deduplication, and encryption. Adding this comprehensive list of features into an existing open-source file system or creating a new feature-rich distributed file system would have been prohibitively expensive in terms of engineering resources and time.

Because these features had been baked into FlexVols over decades, building FlexGroups out of a collection of FlexVols was the obvious and prudent approach. It greatly simplified several aspects of our design. For example, the ability to nondisruptively move FlexVols between nodes is particularly useful for coarse-grained rebalancing without embedding any forward pointers in the file system; for more on this topic, see Sec. 3.3. Moreover, the NVLog, together with the transactional semantics of FlexVols [2, 7, 9, 17, 20, 29] provides the atomicity and reliability for maintaining metadata necessary for the distributed namespace.

More importantly, this choice simplified existing customers' workflows—a FlexGroup looks much like a FlexVol to the administrator and NFS or SMB clients—which was

key to its rapid adoption. FlexGroups also allow "upgrading" an existing FlexVol to a FlexGroup, on-the-fly addition of member FlexVols, presentation of a FlexGroup as a single volume for all administrative tasks, and other management features. Such details are outside of the scope of this paper.

3.1 Design Considerations

Data distribution comes at a cost. Typically, spreading data across nodes requires either a metadata server or internal pointers to redirect from one location to another. Maintaining this metadata costs additional CPU, network, and storage. Coarse-grained distribution of large directory subtrees minimizes metadata costs, but makes it harder to achieve balance between nodes. A distributed system can achieve better balance with a finer-grained distribution, but that comes with the additional cost of more pointers.

Because both fine- and coarse-grained distribution have drawbacks, a promising approach would be to offer a mechanism that can adaptively change distribution granularities based on need—using fine-grained distribution when necessary to ensure the use of all resources, while reverting to coarse-grained distribution when possible for higher overall performance.

An alternative approach to achieve fine-grained distribution without metadata overhead is to use hash-based algorithms to place each file on a pseudo-random node. We ruled this approach out for two reasons. First, hash-based placement causes a large fraction, $(n-1)/n$, of requests to pay the cost of a network hop (NBlade to DBlade) in an n-node system. More intelligent placement of files can reduce hops; for example, files within a directory could be colocated in the common case. Hash-based placement also prevents dynamic placement decisions based on current conditions; for example, avoiding new file creation on a node that is experiencing high load.

Ideally, a system should be capable of reacting to imbalance in load and storage space availability. Retroactive data movement has two serious challenges—picking the content to be moved and doing it in a *nondisruptive* fashion to the clients (i.e., no remounts). The latter requires inserting remote pointers in the file system to avoid invalidating file handles that were previously issued to clients.[1] However, the proliferation of these pointers over time creates an ever-increasing drag on overall system performance. Additionally, although centralizing metadata in one server simplifies some aspects, it leads to obvious performance bottlenecks.

We explored two less-successful designs before FlexGroups. In our first design, all data files were striped (based on file offset and range) across member FlexVols. Subsequently, metadata (directories and inode tables) were also striped and decentralized. A distributed ticketing mechanism was used to maintain cross-FlexVol consistency. However, the resultant fine-grained synchronization generated a high performance tax. In addition, increase in CPU core count in nodes and scaling improvements to the WAFL parallelism model [5] enabled concurrent execution of dozens of read and write operations to a single file, which obviated the need to stripe "hot" files.

The second design stored an inode indirection layer in a master FlexVol, which pointed to data files that were placed in other FlexVols based on simple round-robin policies. The design traded the extra FlexVol hop paid by each operation to consult and follow the indirection for the ease of moving data and metadata across the member FlexVols in response to imbalance in load or storage capacity. However, the design was unable to prevent the master FlexVol (hosting the indirection layer) from becoming a performance bottleneck.

To demonstrate the overhead of this indirection-based scheme, we benchmarked it against FlexGroups, which, as explained later in this section, have several important differences. FlexGroups use pointers rather than indirection to locate remote files and try to place each file in the same node as its parent directory so that most operations avoid an extra NBlade-to-DBlade hop. Finally, FlexGroup members are symmetric; no node or FlexVol necessarily handles more (or less) data, metadata, or traffic than its peers.

We ran SPEC SFS 2014 SWBUILD [35] on both a FlexGroup and the indirection-based approach using a mid-range HA pair.[2] The operational throughput (ops/s) of FlexGroups was more than 4 times that of the indirection approach. At low load points, both were easily able to handle the workload, but the average request latency of the indirection approach was 1.9 times higher than that of FlexGroups, reflecting the extra network hop required by every request.

This pattern continued at higher load points until the master FlexVol became a bottleneck in the indirection-based system. Queue lengths grew rapidly with increasing load, until a 20-fold increase in time spent to resolve indirection which caused the operational throughput to collapse in spite of the fact that almost half of the available CPU cores remained idle. In contrast, FlexGroup performance continued to scale until the cores on both nodes were almost fully utilized. Note that this test was performed on a 2-node cluster. The indirection bottleneck is more acute in a larger cluster.

3.2 Fusing FlexVols via Remote Hardlinks

FlexGroups distribute both data and metadata across multiple FlexVols by allowing directory entries to be *remote hardlinks* to inodes on other member FlexVols, as illustrated in Fig. 2. A client can connect to any node in the cluster, and the NBlade routes its requests to the appropriate FlexVol and DBlade.

FlexGroups perform most data distribution during *ingest*: an intelligent, immediate, and permanent placement decision

[1] NFS requires long-lived file handles.

[2] Each node had 16 cores and 96 GiB of DRAM.

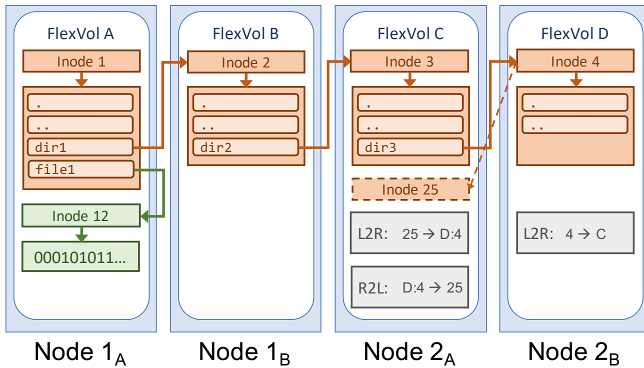


Figure 2: An example of a FlexGroup (DBlades only) that consists of member FlexVols in four nodes, along with hardlinks following `/dir1/dir2/dir3`. L2R and R2L are local-to-remote and remote-to-local databases. FlexVol C contains inode 25, which is a cached version of inode 4 from FlexVol D.

is taken when creating a new file or directory, thereby assigning the new object to one of the FlexVol members of the FlexGroup. FlexGroups adjust the granularity of distribution from the directory level down to individual files as necessary.

As explained in Sec. 3.1, striping individual files was not necessary to achieve our performance goals. Therefore, FlexGroups do not stripe individual files or directories across member FlexVols; each object lives entirely in one FlexVol. This greatly simplifies the design and allows all *data* operations (reads and writes) to be executed with the *same latency and throughput*. In other words, remote hardlinks are not traversed when servicing such operations; an additional network hop is incurred only if the NBlade connected to the client and DBlade in which the data resides are on different nodes. Only *metadata* operations that resolve or modify remote hardlinks incur additional latency for the extra internode communication.

The FlexVol identifier and the inode identifier are encoded within the opaque file handle that WAFL returns to operations such as `LOOKUP` and `OPEN`. An NFS client resolves `/dir1/dir2/foo` by starting at the root located in FlexVol A. Suppose that the client connects to Node 1_A and sends a `LOOKUP` operation for `dir1` to that NBlade. This operation is directed to the same node's DBlade, which contains the root directory, and the remote hardlink for `dir1` is located. The NFS handle returned to the client has B:2 encoded within it. The client then sends a `LOOKUP` for `dir2` using this opaque handle, which the NBlade forwards to FlexVol B (Node 1_B). The NFS handle returned to the client now embeds C:3. A `LOOKUP` for file `foo` returns a file handle that encodes the FlexVol (which could even be A) and the inode for that file. All subsequent reads and writes on that file get routed by the NBlade of Node 1_A directly to the correct DBlade with no further access of the remote hardlinks. Thus, as mentioned earlier, every data operation costs the same and is independent of the number of remote hardlinks that led to it.

Unlike NFS, the SMB client sends the entire pathname, starting from the root. For example, an SMB client may `OPEN /dir1/dir2/foo`, whereas an NFS client sends a series of `LOOKUP` operations that walk down to `foo`. NBlades accelerate these SMB operations by maintaining in-memory pathname-to-FlexVol tables that can be used to resolve an entire pathname to a FlexVol, similar to Sprite prefix tables [41]. Depending on the state of the table, the `OPEN` operation may avoid resolving some or all remote hardlinks in the pathname when it gets routed to a FlexVol. The resolved hardlinks are added to this table. Much as with NFS, once `OPEN`ed all subsequent reads and writes to that file are routed directly to the correct DBlade.

3.3 Load Balancing

Based on experience with prior designs and customer workloads we realized that remote hardlinks with ingest-based placement yield a close-to-optimal trade-off by reducing the necessary synchronization overhead even while reducing the likelihood of imbalances in storage capacity and load across member FlexVols. Sec. 3.5 explains how placement heuristics are based on both the recent IOPS load imbalance and the capacity imbalance across member FlexVols. Obviously, these heuristics cannot predictably avoid load imbalances in the future. For example, it is possible that data written aptly at ingest time to one (or a few) member FlexVols becomes "hot" several hours or days later due to some application workflow. As explained earlier, nondisruptive, retroactive, and fine-grained data redistribution requires creating permanent remote markers that will regress the overall file system performance over time. Instead, such a pathologically imbalanced FlexGroup is fixed efficiently and expeditiously by coarse-grained movement of "hot" member FlexVols to nodes that are sustaining less IOPS load. This is done by leveraging the *Volume Move* feature [26], which leaves no residual remote markers in the file system namespace of the FlexGroup.[3]

3.4 The Remote Access Layer

FlexGroup volumes work by having one member FlexVol take responsibility for executing an entire operation, updating its own state, and coordinating the state changes with the other members. Each time an operation needs to create, delete, or access a remote hardlink, control passes to the *Remote Access Layer (RAL)*. The RAL is responsible for managing and updating remote hardlinks in a transactional

[3]Movement is accomplished by taking periodic snapshots of the FlexVol and incrementally transferring all changes to the destination node. The frequency of these transfers increases as the movement catches up with the most recent version of the FlexVol at which point the volume information database consulted by the NBlades is atomically updated.

manner and allowing the existing file system code to operate on remote metadata as if it were local. The RAL is also responsible for recovering hardlink state after crashes and for dealing with network slowness and outages between nodes.

Conceptually, the RAL is a delegation service. When an operation accesses a remote hardlink, the RAL requests the corresponding metadata from the remote FlexVol and places it in a metadata cache where it can be accessed by the local operation. This caching represents a delegation from an *origin* FlexVol to the *caching* FlexVol. A delegation may be released proactively by the caching FlexVol or revoked by the origin FlexVol. Because the objects being cached are files or directories, each cache entry is conveniently stored in an inode at the caching FlexVol; the inode is freed when the caching relationship (and delegation) cease to exist.

The RAL persists its cached metadata to ensure that it is not lost in the event of a node failure. As described in Sec. 2, when a node fails, its HA partner takes over all of its FlexVols, replays logged operations, and resumes service of those FlexVols almost immediately. The RAL metadata caches benefit from this same high availability by persisting their state to metadata files in the FlexVols. This makes delegations fault tolerant and therefore simplifies the RAL design.

There are other approaches for tracking and maintaining distributed state, such as remote hardlinks. Porting and leveraging well-known services, such as ZooKeeper [15] or etcd [4], to maintain this state was tempting for reasons such as speed of implementation. However, building the RAL transactional mechanisms within our file system provided two major advantages: (1) low transactional overhead, because RAL bookkeeping occurs in the context of the client operation; and (2) resilient transaction tracking, because RAL bookkeeping leverages the enterprise-quality transactional semantics provided by WAFL.

The next three subsections explain the transactional infrastructure for creating and managing remote hardlinks.

3.4.1 RAL Caches

As summarized above, the RAL is a write-back metadata cache of remote inodes. The caching FlexVol creates (if necessary) and uses delegations cached in local inodes from one or more origin FlexVols to execute file system operations. These inodes may contain read-only (RO) or read-write (RW) caches. In other words, for any given object, its origin FlexVol can grant at most one exclusive RW cache to another FlexVol or multiple RO caches to multiple FlexVols.

The cached inodes are stored persistently in the regular inode table and are used by local file system operations much like regular inodes. Because they are stored in the local file system, updates to cached inodes are protected by the NVLog. This ensures that the delegations represented by cache entries are not lost in the event of node failures.

The combination of HA pairs with the mirrored NVLog, as described in Sec. 2, ensures that node failures appear, at worst, as transient delays to the RAL. When a node fails, its partner quickly takes over, replaying operations from the NVLog to recreate the RAL metadata state that had not yet been persisted to storage.

The caches are managed using *local-to-remote* (L2R) and *remote-to-local* (R2L) maps, which are implemented as B+ trees and stored in each FlexVol as hidden files. The L2R maps store two kinds of information: (1) the map from locally cached inodes to the corresponding remote inode number and origin FlexVol ID; and (2) the map of local inodes that are cached by other FlexVols. The R2L maps store the reverse mapping of the first type of L2R map data. For example, in Fig. 2, inode 25 on FlexVol C is a cached version of inode 4 on FlexVol D.

RAL caches are not intended as long-term storage, and are primarily used as (1) a temporary measure to complete metadata operations that choose to locate newly created content in a remote member; and (2) a mechanism for providing crash-consistent transactional semantics.

Caches can be evicted proactively by the caching FlexVol or revoked by the origin FlexVol. A background *scrub* process periodically walks and reclaims all caches, because caches are only needed temporarily. In fact, to improve performance, most read-write caches are aggressively evicted after their use, so that they don't have to be evicted when a later operation needs to create a cache of the inode in a different FlexVol. The scrub also reclaims all wasteful or stale outstanding references. The origin FlexVol can evict read-write caches of its inodes by examining its L2R map to find the set of caching FlexVols, sending them `REMOTE_WRITEBACK` messages to write back the dirty data, and then evicting the entries.

3.4.2 Example

We use an NFS `MKDIR` request—a sufficiently complex operation—to illustrate how RAL is used. Suppose that subdirectory `dir3` is created under `dir2`, that `dir2` is stored on FlexVol C, and that the placement logic decides to store the contents of `dir3` on FlexVol D. Fig. 2 illustrates this example.

1. FlexVol C suspends local processing of the `MKDIR` operation when it determines that it needs a remote hardlink.
2. FlexVol C sends a `RAL_RETRIEVE` message to FlexVol D, asking for the creation of a new inode and a copy of it for read-write caching.
3. FlexVol D allocates the new inode (4 in the figure), tags it as being cached elsewhere in the RW manner, and creates the corresponding L2R entry.
4. FlexVol D responds to FlexVol C by sending a `RAL_STORE` message to store the cached copy of this inode in FlexVol C.

5. FlexVol C processes the RAL_STORE message by allocating a local inode (25 in the figure), tags it as a RW cache, creates the appropriate L2R and R2L entries, and adds it to the pool of cached inodes for FlexVol D.
6. FlexVol C restarts the local MKDIR operation.

The MKDIR operation finds the cached inode in the pool and proceeds to completion using it as the proxy for dir3[4]. The MKDIR operation now creates the remote hardlink directory entry in dir2 pointing to dir3, converts the inode C:25 into a directory type, populates the . and .. entries in it, and marks the RW cache entry as "dirty."

If the client now sends a CREATE command to create a new file in dir3, that request gets routed to FlexVol D. When the operation executes, it notices the tag on dir3's inode that marks it as being in an RW cache elsewhere. The FlexVol suspends the operation, consults the L2R map to determine that it is cached in FlexVol C, and starts the eviction process. FlexVol C writes back the dirty RW cache to FlexVol D, removes it from the cache, and FlexVol D then marks the inode as a valid subdirectory with no RW caches. The CREATE operation is then resumed in FlexVol D.

As the MKDIR example illustrates, the DBlade of the caching FlexVol takes ownership of executing the client operation atomically, but only after it has created local RW caches from an origin FlexVol. We now briefly describe an example RENAME operation, which is more complex and may involve two origin FlexVols. A RENAME mv dirA/foo dirB/bar operation is routed to the DBlade that hosts dirB. Suppose that dirA is in FlexVol A, dirB is in FlexVol B, and foo is a hardlink from dirA to inodeC in FlexVol C. FlexVol B first acquires two RW caches[5]—one from FlexVol A for dirA and one from FlexVol C for inodeC—and then executes the RENAME as a regular WAFL operation that uses the two RW caches. The execution of the RENAME deletes bar if one already exists, creates a new local bar that hardlinks to inodeC, and deletes foo in the RW cache for dirA. The persistent RW cache entries are left in a "dirty" state; the origin FlexVols are modified when these cache entries are eventually flushed back.

Some operations require read-only (RO) caches; for example, an SMB OPEN operation creates RO caches for any traversed remote inodes while resolving a file pathname.

3.4.3 Consistency and Recovery

The FlexGroup consistency model builds on the transactional semantics of the underlying COW WAFL file system with its NVLog. A client operation is acknowledged only after it has been recorded in the NVLog of both the node and its HA partner. All RAL operations are similarly recorded to NVLog before they are acknowledged. After a node failure, NVLog replay returns the affected member FlexVols to a state representing a valid phase of any ongoing RAL transaction. It must be noted that the ONTAP HA-pair model (together with our RAID and WAFL software) is specifically engineered to be highly reliable[6], which informs the rest of this section.

For example, in the case of the MKDIR example, the node containing FlexVol D processes the RAL_RETRIEVE message, records the message and result in its NVLog, and responds to FlexVol C. If FlexVol D's node fails before the corresponding updates are persisted to the local file system, its HA partner replays the RAL_RETRIEVE message, recreating the inode.

If the node that hosts FlexVol C fails during the RAL_STORE phase, its HA partner may choose a different location for creating dir3 when it replays MKDIR. FlexVols D and C may be left with stale or unused cache entries or temporary inodes. This is the only allowed form of inconsistency after a node failure; it is eventually resolved by the background scrub process, which periodically walks and reclaims all caches, including all wasteful or stale outstanding references[7]. The continued existence of these stale references is safe and does not compromise the consistency of the file system.

Any dirty state left in the cache inode (25 in the MKDIR example) is recreated by replaying MKDIR if C fails after MKDIR is logged but before the subsequent file system transaction completes. Although a formal proof is not provided for lack of space, we conclude that *RAL usage for FlexGroup is crash consistent*.

Sec. 2 explains that no fsck-style repair is required if a node fails or even in the rare case that NVLog in both nodes of an HA pair is lost. When a member node in a FlexGroup fails, its HA partner replays its NVLog and recreates the file system state, including the RAL state. However, the loss of NVLog of one member FlexVol may result in inconsistencies between it and other members. Additional mechanisms were built to accomplish automatic on-the-fly repair when such an inconsistency is detected by an operation. In brief, the operation is suspended while a high-priority WAFL message investigates the inconsistency, fixes it, syslogs it, and restarts the original operation. For example, suppose that a LOOKUP consults a directory entry foo, which is a remote hardlink to an inode in FlexVol B that does not exist because FlexVol B suffered a failure followed by a loss of its NVLog. On-the-fly repair would be kicked off once the RAL_RETRIEVE fails,

[4] In theory, the MKDIR operation could find a RW cache entry in that pool on its first execution. That RW cache entry might have been created due to a RAL_RETRIEVE operation to member D initiated by a different operation. In that case, on resumption that operation kicks off yet another RAL_RETRIEVE operation to populate the pool.

[5] Each cache is acquired using the RAL_RETRIEVE and RAL_STORE handshake.

[6] The availability of our customers' systems is routinely measured at 5 to 6 nines; that is, annual system downtime between 3 and 30 seconds.

[7] The WAFL file system includes a time-tested background *scanner* infrastructure used to walk and operate on various file system metadata. The infrastructure paces itself based on current system load, thereby ensuring negligible (less than 2%) impact to client operations. There are about 20 different *scan* types, and the FlexGroup scrub is one of them.

which would eventually delete the entry `foo`. The restarted `LOOKUP` now finds no corresponding entry. A detailed discussion of on-the-fly repair techniques is beyond the scope of this paper.

3.5 Ingest Heuristics

As mentioned earlier, all placement decisions are made during ingest before the new inode is allocated. The heuristics balance two competing goals: distributing load (IOPS) and capacity among member FlexVols versus reducing the operational overhead associated with remote hardlinks.

Creating a remote hardlink to an idle member almost immediately brings traffic to it, which shifts some traffic to idle members and increases overall performance. Additionally, creating a remote hardlink to an underutilized FlexVol almost certainly causes it to fill up a little more, which helps bring its usage in line with that of its peers. This is more important when the member volumes are closer to full, and remote hardlinks can help get to every last byte of storage. Therefore, as the FlexVols become more and more full, the FlexGroup should employ more remote hardlinks.

On the other hand, every time the FlexGroup creates a new remote hardlink, there is a small performance penalty, both at the time of creation and in the future when accessing the remote hardlink. The penalties accumulated across too many remote hardlinks increase average request latency and reduce overall performance. However, too few remote hardlinks may mean a failure to use all available resources. Thus, the primary goal of ingest heuristics is to achieve the right balance while using as few remote hardlinks as possible.

3.5.1 Input to Heuristics

To minimize the overhead of heuristics, each node makes independent allocation decisions based on the following information about each member FlexVol:

- *Block usage:* The ratio of consumed to total storage space of the FlexVol.
- *Inode usage:* The ratio of the number of locally allocated inodes to the maximum number of inodes allowed for the FlexVol.
- *Recent ingest load:* The recent frequency with which it has created new files and directories (maintained by using a sliding window average across several seconds)[8].

The member FlexVols periodically exchange this information by using an asynchronous and lightweight viral *refresh process* provided by ONTAP. Each node propagates information about its local member FlexVols as well as information received from other nodes. Each node sends out a message to each peer every second and uses timestamps to decide the staleness of the received information. An asynchronous model suffices because the heuristics are reacting to trends in load and storage consumption rather than making instantaneously optimal decisions.

3.5.2 Heuristic Probability Tables

Each node consults a set of probability tables during inode allocation. The tables are recomputed approximately every second, using any new information that is received from the refresh process. The table in each FlexVol consists of two arrays:

- $RP[c]$: An array of *remote preference*—a value between 0 and 1 indicating the probability of remote allocation for objects of various categories, such as files and subdirectories at various depths from the root directory.
- $AT[m]$: An array of remote *allocation target* probability values for member FlexVol m; the sum of values across the array is 1.

When making an inode allocation decision, the FlexVol generates a random number between 0 and 1 and compares it to the corresponding value in RP for that category. If the randomly generated number is larger, it processes the allocation locally; otherwise it uses a weighted round-robin scheme to determine the FlexVol for remote allocation: Each remote member is assigned a target percentage based on its AT value, and each allocation selects the remote FlexVol that is furthest behind its target percentage compared to its peers.

3.5.3 Computing Heuristic Probability Tables

The probability tables are recomputed by comparing the properties of all member FlexVols and looking for trends and problem conditions. As mentioned earlier, each node computes these tables independently based on the information from the most recent refresh process. First, two intermediate values—*Urgency* and *Tolerance*—are computed.

Urgency biases the heuristics to react to imbalance in the FlexGroup. It is computed as a linear interpolation of each member's usage and each node's ingest load within their respective high and low thresholds, which are precomputed based on the FlexGroup configuration. An *Urgency* of 1 indicates that at least one member volume is critically low on one of these resources. Values between 0 and 1 indicate escalating degrees of concern.

Tolerance indicates how much disparity in load or usage among member volumes is acceptable. A low value tells the heuristics to react more strongly to disparities between member FlexVols. A FlexGroup that is empty with no load will have maximum *Tolerance*. As a FlexGroup gets closer to full capacity, *Tolerance* goes down, and the heuristics allow less disparity among the members.

[8] Sec. 3.3 explains the handling of unpredictable future imbalance in load across the member FlexVols.

Computing Remote Allocation Target (AT) Probabilities: First a hypothetical usage goal is computed—somewhere between the highest current capacity usage (combining both blocks and inodes) on any member and the maximum capacity of any member. The heuristics assign each member an allocation target based on the difference between that member's usage goal and its current usage. In essence, the heuristics select targets such that if all allocations were remote and all new files were exactly the same size, the remote members would then fill up at exactly the rates needed to reach their usage goals at the same time.

A non-zero *Urgency* affects this calculation significantly. A member that contributes to the non-zero *Urgency* is given a much lower target. For example, a member with the maximum *Urgency* value of 1 is assigned only 1% of its target. Once target values are assigned to each member, the values are normalized into $AT[m]$ as probabilities summing to 1.

Computing Remote Preference (RP) Probabilities: The heuristics iterate over each allocation category for a member FlexVol. Some categories are easily computed. For example, allocating a new subdirectory in the root directory is always a remote allocation to ensure that this new branch of content lands on the member with the least capacity usage or load. But for most categories, the calculation is more complex and uses the recent ingest load data from all members.

First, the heuristics compare the recent request load for an allocation category to the target load specified by the volume's allocation target (AT). If the recent load is below the target, then $RP[c]$ is set to 0, indicating a desire for the FlexVol to satisfy new allocations locally. However, if the recent load is above the target then $RP[c]$ is computed as the proportion of the load that is in excess of the target. For example, if a member with a target of 8% has recently received 10% of the overall allocations in a category, then that category is assigned a *RP* of 0.2 so it can attain its target. As an optimization, *RP* is reduced for members that have exceeded their target by less than the current *Tolerance* value, optimistically allowing them to keep a higher percentage of local traffic for local placement.

Again, a non-zero *Urgency* value for a member increases its *RP* values. As a member FlexVol or node runs low on resources, the allocations are more likely to happen on remote peers.

4 Topics in Practice

Building a scale-out file system required meeting customer expectations of the features, performance, and robustness that they were accustomed to with FlexVols. This section touches on a few selected topics related to satisfying those expectations.

Snapshots: Several features of ONTAP depend on the ability to efficiently create consistent snapshots. A FlexGroup snapshot is a collection of snapshots, one per member FlexVol, that are created in a coordinated fashion. First, each member FlexVol fences all new client operations and evicts all RW caches. Then each member FlexVol independently creates a snapshot and drops its fence. Because very few RW caches are outstanding at any point in time, this fencing creates no noticeable disruption to client performance (both latency and throughput of client operations). The design choice to evict all RW caches was made to avoid extra implementation work in various internal file system operations to understand, handle, and traverse RAL information when accessing snapshots. Eviction is not really necessary because the L2R and R2L metadata is consistently captured in each member FlexVol snapshot, and could be used to service reads of the FlexGroup snapshot image. The metadata can also be reactivated in the case of a restore of the entire FlexGroup to that coordinated image.

Quotas: Tracking and enforcement of user, directory, and group quotas must treat the entire FlexGroup as a single entity. Any incoming operation must fail when a quota rule is violated. Caches created by the RAL infrastructure count toward quota consumption. Quota credits are pro-actively distributed across member FlexVols to allow efficient, independent, and per-operation granular enforcement of the rules. In the worst case, an operation may be suspended while the FlexVol communicates with other members to borrow credits; the design makes such scenarios extremely rare.

Unreachable Member FlexVols: One or more members may become temporarily unreachable; for example, due to network problems. All client and RAL operations directed to those FlexVols will time out and get retried. Meanwhile, access to data in other FlexVols continues as usual. ONTAP clustering services indicate whether the FlexVols still exist, whether the outage is temporary, and whether retries will eventually succeed. If the problem is not transient or if clustering services indicate that FlexVols have been destroyed, either the FlexGroup can be restored to its most recent coordinated snapshot[9] or on-the-fly repair will eventually fix RAL metadata that point to the lost FlexVols. Both approaches recover file system consistency but result in data loss. The former is typically preferred because the loss is recent and predictable; all mutations after the snapshot are lost.

Testing: Enterprise-grade quality implies continuous validation; 102 different test suites are executed, totaling 160 hours of runtime daily. These tests use both NFS and SMB clients to specifically stress cross-member code paths that use RAL. Many of the suites also inject errors, such as dropping RAL operations, forcing node panics to trigger HA events, discarding NVLog during HA-events, and artificially creating memory pressure. There are also suites that explicitly create inconsistencies in the persisted RAL metadata to test on-the-fly repair mechanisms that correct them.

[9]Snapshots can be replicated to and restored from remote nodes by using NetApp SnapMirror® [29].

5 Evaluation

This section shows that the load-balancing automation of FlexGroups compares well to an ideal FlexVol in three areas: overhead, scale, and balance. It is not practical to formulate an apples-to-apples comparison of FlexGroup to other well-known distributed file systems, due to the difference in configurations, sizes, and associated feature sets. Instead, this section compares FlexGroup performance to ideal and worst-case scenarios that are manually configured (as explained below). Experiments to measure FlexGroup overhead (Sec. 5.1) and scale (Sec. 5.2) were completed using a cluster of up to 8 nodes, each with two 6-core Intel Broadwell DE processors, 64 GiB of DRAM, and a shelf of 24 SSDs. Capacity balancing was validated with data collected from customer deployments (Sec. 5.3).

5.1 Overhead

Automatic redirection of files and directories between FlexVols in a manner that is consistent in the face of faults adds two major sources of overhead: RAL (Sec. 3.4) introduces additional overhead for metadata operations, and some operations incur an additional network communication cost when the client sends a request to an NBlade that cannot be satisfied by the local DBlade. We measure these overheads by comparing the performance of FlexGroups to two manually created configurations with FlexVols, neither of which incur the overhead of RAL: (1) **FlexVol-Local** is an ideal configuration in which operations are routed by each NBlade to the DBlade on the same node. (2) **FlexVol-Remote** is a configuration in which operations are always routed by each NBlade to a DBlade on a different node.

Even though we used a single HA pair in these experiments, the results in these two cases are independent of the number of nodes: Either none (FlexVol-Local) or all (FlexVol-Remote) operations involve inter-node processing, and this does not change with the number of nodes. In the case of FlexGroups, the remote-to-local ratio might increase with the number of nodes, but no other are latencies added.

5.1.1 Overhead of NFS Operations

We measured overhead by generating a metadata load using `mdtest` [14] to a single HA pair. We used a single mdtest client connected by using NFSv3 to one of the nodes, with no other load on the system. We report the latencies of the individual metadata operations, measured on the storage systems. We configured mdtest to create approximately 2 million directories and 2 million 256KB files; the maximum NFS transfer size was set to 64KB.

Fig. 3 shows the normalized results. We observe that in the FlexGroup case, data is spread approximately evenly across the two DBlades. Read-only metadata operations that use NFS file handles, namely `ACCESS` and `GETATTR`, exhibit performance that is almost exactly halfway between the local and remote cases because there is no RAL overhead. Satisfying the request incurs additional communication across the cluster interconnect approximately 50% of the time. The `LOOKUP` operation incurs additional latency when resolving a name in a directory that happens to be a remote hardlink, because it creates a RO cache of the looked-up inode on the same node as its parent directory[10]. Metadata update operations, such as `CREATE`, `MKDIR`, `REMOVE`, and `RMDIR`, show the overhead of RAL and of communicating over the cluster interconnect, each occurring roughly 50% of the time.

Even though many of these metadata operations incur an overhead, they are relatively infrequent compared to data operations. Read operations exhibit performance that is approximately halfway between the local and remote cases. Write operations perform comparably worse than read operations because they require updating file inodes; for example, to extend the file lengths and to update `mtime`. As shown later in this section, the overall impact on performance is minimal when looking at the operations in aggregate.

5.1.2 Application and Data Benchmarks

Fig. 4 shows the performance of our application benchmarks, expressed as normalized operations per second. (Higher is better.) The figure presents two sets of results: random and sequential read and write benchmarks, and selected SPEC SFS 2014 benchmarks [35], which evaluate a realistic mixture of file system operations.

We generated the random and sequential results for reads and writes by using an internal benchmark that increases the load to find the maximum throughput possible while placing data in accordance with the configuration (FlexVol-Local, FlexVol-Remote, or FlexGroup). Unlike in the mdtest experiment, in which only one NBlade and one DBlade were active at any given time, in the FlexVol-Local case, we used several clients to saturate both nodes with requests. The results indicate that throughput for FlexGroups achieves a balance somewhere between the best (FlexVol-Local) and worst (FlexVol-Remote) cases.

SFS [35] is a standard file system workload generation tool that comes with profiles generated from real-world examples. We use three SFS profiles representing different mixes of metadata requests and data throughput [37]: **SWBUILD,** heavy metadata similar to Linux kernel builds; **EDA,** a balance of metadata and data throughput representative of electronic design automation applications; and **VDA,** streaming writes and few metadata operations, similar to a video recording system. Our goal in these benchmarks is to determine peak operations/second, not to produce compliant SFS numbers as defined by SPEC [36].

[10] The performance of `LOOKUP` is independent of the path length because NFSv3 resolves each pathname component separately.

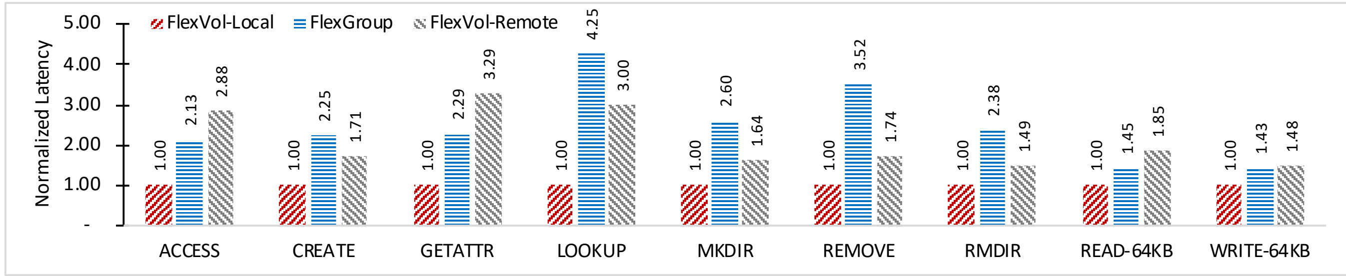


Figure 3: Server-side latency for select NFS operations generated by a single mdtest client, reported relative to FlexVol-Local latencies. Lower scores indicate better performance.

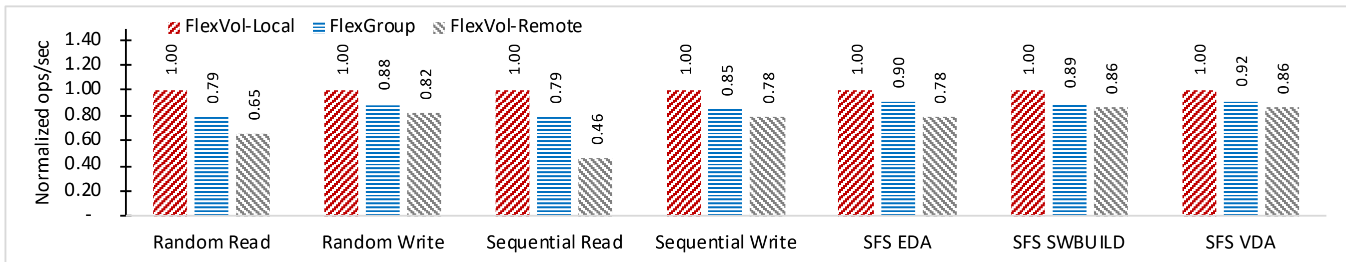


Figure 4: Operations per second for application and data benchmarksrelative to FlexVol-Local. Higher scores indicate better performance.

For the EDA and VDA workloads, FlexGroup performance fell almost exactly between FlexVol-Local and FlexVol-Remote, showing that the load-balancing heuristics achieved a balanced distribution of files across the two test nodes and that there was negligible overhead from RAL. In the SWBUILD test, FlexGroup performance was again between the extremes but closer to FlexVol-Remote. In this test, 87% of requests are metadata operations, increasing overhead due to more frequent RAL processing.

We also determined via profiling that the recomputation of the heuristic probability tables every second by each node adds a negligible amount of CPU cycles (less than 0.001%), even for large customer deployments.

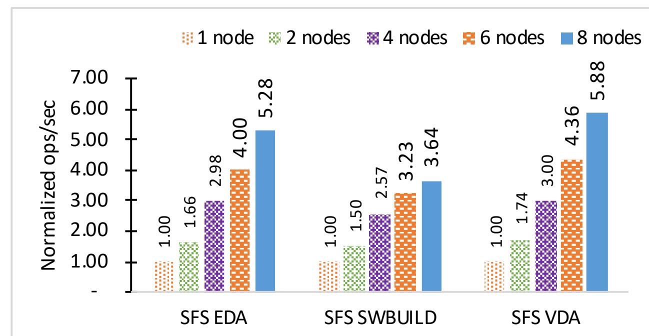


Figure 5: Scalability of FlexGroups. The max achieved operations/second for three SFS profiles, run against 1, 2, 4, 6, and 8 nodes. Each metric is reported relative to a 1-node cluster. Higher scores indicate better performance.

5.2 Workload Scaling

To examine the FlexGroup scalability under different workloads, we ran the three SFS profiles against 1-, 2-, 4-, 6-, and 8-node clusters (as described at the beginning of Sec. 5). Fig. 5 shows the scaled results. Workloads with lower metadata intensity (VDA) scale better than workloads with more metadata operations, as expected due to the added transaction overhead associated with RAL, discussed in Sec. 3.4.

5.3 Customer Experience

FlexGroups became available to customers in early 2017. In this section, we share information about how customers have used FlexGroups and about improvements made to our heuristics based on that experience. The storage systems can "phone home" to report customer configuration and event data [23]. This functionality is optional, but a large fraction of our customers enable it. Because our customers typically configure multiple FlexGroups and other FlexVols on the same nodes of a cluster, run dozens of applications on any given node of the cluster at various times, and ONTAP does not fully gather IOPS statistics on a per-FlexVol basis, it is not possible to compute or to clearly show whether the IOPS load stays balanced across member FlexVols over a long period of time. On the other hand, per-FlexVol capacity consumption is tracked for imbalance, and it also serves as a good proxy for imbalance in load.

This data shows a steady rise in FlexGroup adoption in

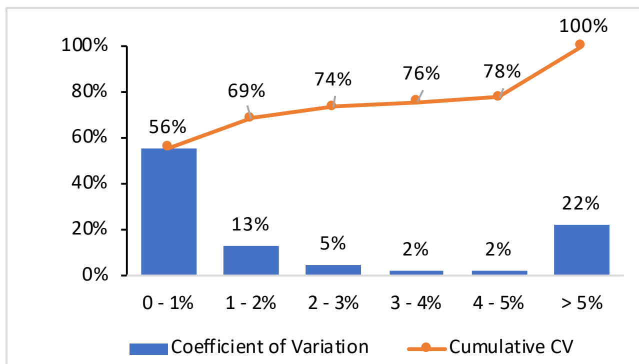


Figure 6: Do the placement algorithms balance data in the real world? This histogram shows the dispersion of FlexVol usage in customer-deployed FlexGroups using the coefficient of variation (CV). CV is the standard deviation normalized by dividing by the mean (σ/μ).A smaller CV indicates lower dispersion. Only FlexGroups greater than 10TB in size and more than 5% utilized were included.

the 2 years since its release. As of August 2018, hundreds of customers have deployed thousands of FlexGroups to manage hundreds of petabytes of storage. Roughly half of these FlexGroups are small (< 10TB), and we surmise that they are being used for testing and evaluation. 25% of these FlexGroups are larger than 100TB, and 5% are larger than 1PB. A handful of FlexGroups are larger than 5PB. Of the FlexGroups that are 10TB or larger, most (70%) have between 8 and 32 member FlexVols. 5% are larger, with the largest containing over 150 members.

Customer data also provides insight into the effectiveness of our ingest heuristics at balancing capacity across the members of a FlexGroup. Standard deviation (stdev or σ) measures dispersion. However, because different FlexGroups contain different amounts of data, the standard deviations are not directly comparable. For example, a stdev of 10^5 bytes would be interpreted differently for a 1TB FlexGroup than for a 100TB one. To normalize these numbers for comparison, we divide the stdev by the mean (μ), producing a coefficient of variation (CV), or σ/μ, that gives the stdev as a percentage of the mean.[11] A FlexGroup with a CV of 1% suggests that each FlexVol has a 95% probability of being $\pm 0.02\mu$; a CV of 3% indicates a 95% probability of being $\pm 0.06\mu$.

The data indicate that the FlexGroup placement algorithms are working in most customer use cases, as shown in Fig. 6. Over half (56%) of FlexGroups had a CV less than 1%, 78% of FlexGroups had a CV less than 5%, and 85% were below 10%. Only FlexGroups larger than 10TB and more than 5% utilized were included in this analysis. For the 15% of FlexGroups that had a CV greater than 10%, we found three patterns that account for most of the cases.

[11] CV is also known as relative standard deviation.

First there is a group of FlexGroups that have a bi-modal usage pattern—one set of members with similar high usage and another set with similar, but lower, usage. These appear to be FlexGroups that customers have expanded by adding a set of new member FlexVols, and the newer members show lower usage than the older ones.

The second pattern includes FlexGroups that hold a small number of very large files. An example would be a 200 TB FlexGroup with a small number of backup archives averaging 100 GB each. In these cases there aren't enough files to average out the effects of our ingest decisions, and the impact of allocating (extremely large) outlier file sizes can increase the CV of the FlexGroup.

Finally, there is a small number of FlexGroups that are well balanced except for a single volume with much higher usage. We believe that these cases are caused by oddities in customer workloads: an output archive file that encompasses an enormous amount of workload data, or a directory of log files that were co-located when they were created, but have grown very large over time.

It should be noted that such examples of capacity imbalance do not necessarily imply imbalance in IOPS load across the member FlexVols. In the second pattern, backup files experience sequential appends and infrequent sequential reads. In the third pattern, output archive files experience infrequent bursts of append operations. The WAFL file system is well tuned to absorb a spike in reads and writes to a single FlexVol. And as mentioned earlier, in the worst case a Volume Move operation can relocate such a FlexVol from a node that happens to be overloaded.

Based on specific customer experiences, we have improved the ingest heuristics over the four releases since FlexGroups were introduced. Early on, an interesting customer case motivated the addition of the inode usage property to the refresh process. Without accounting for inode usage, the heuristics had kept several million small files in a few members to balance out some extremely large files in the others. The per-FlexVol limit on the number of inodes was hit, causing out-of-space errors. Another customer experience resulted in converting the ingest load into a sliding window average to accommodate spikes. Other tweaks were made to various constants used while recomputing the probability tables to prevent the heuristics from over-reacting to changes.

5.4 Applicability Beyond WAFL

The FlexGroup design builds on years of engineering investment in our WAFL file system, but could these concepts be applied to other file systems? FlexGroups required modest changes to WAFL, and we believe that similar enhancements are possible to other file systems: remote hardlinks and the ability to traverse, create, and delete them. The inputs to ingest heuristics are simple and should be easy to implement.

The biggest challenges may be availability and fault tol-

erance. A robust file system like WAFL can persist RAL metadata locally with its reliable native consistency semantics. Other file systems may require more expensive distributed consensus techniques, like two-phase commit [22] or Paxos [21], to ensure fault tolerant updates to remote hardlinks.

6 Related Work

Many distributed file systems have been developed by the research, commercial, and open source communities. To discuss FlexGroups in the context of this large body of prior work, we focus on file systems that share our design goals and implementation choices. Thus we emphasize systems in which storage devices are controlled by a single node (or two nodes for fault tolerance) and in which nodes manage data at the granularity of files or objects. We will not discuss shared disk file systems such as GPFS [32] or Frangipani [38].

Distributed file systems use a variety of strategies for assigning files and directories to nodes. The simplest approach is a static partitioning, whereby an administrator assigns namespace subtrees to different servers. This strategy is exemplified by systems like NFS [31], AFS [13], and Sprite [27]. It was also the approach that ONTAP supported prior to the introduction of FlexGroups [8].

The disadvantage of static namespace partitioning is uneven load and capacity balancing. Dynamic distribution addresses this challenge by selecting or updating file locations on the fly. Slice [1] compared two heuristics for dynamic partitioning. *Name hashing* maximizes balancing by assigning every new file and directory to a random node chosen by hashing its name. *Mkdir switching* maintains namespace locality by assigning a configurable fraction of new directories to a different node than their parent and allocating all other files and directories on the same node as their parent.

The name hashing strategy has also been used in other distributed file systems including Vesta [3] and GlusterFS [12]. Like mkdir switching, FlexGroups aim to maintain namespace locality by rarely using remote links. But FlexGroups also uses current load and capacity in deciding when to split the namespace.

Unlike FlexGroups, many distributed file systems separate namespace management from data storage. These designs have nodes that store objects and a metadata service that maps filenames to objects. For scalability, some systems have multiple metadata servers, introducing the same tradeoff between load balancing and namespace locality that we address in FlexGroups. Ceph [39] is a widely-used system of this type. Unlike FlexGroups, Ceph repartitions the namespace in response to observed load. This is facilitated by Ceph's use of separate metadata servers; migrating a namespace subtree does not require moving the corresponding data objects. Policies for migrating subtrees is an area of ongoing research [34]. Ceph manages data placement using a hash-based algorithm to select object storage devices [40].

PanFS [42] represents another point in the design space. It statically partitions its namespace across metadata managers and randomly places files on different object servers (blades). Like FlexGroups it adjusts its allocation probabilities to reflect disparities in free space across the object servers. PanFS can also actively balance capacity by relocating data objects. Unlike FlexGroups, PanFS does not take load into account during data placement.

Farsite [6] takes a unique approach to metadata partitioning. It spreads files across servers based on file identifiers. But instead of using a hash, it uses a tree-structured system of file identifiers. This supports the colocation of related files, while avoiding the problem of directory renames forcing data migration between servers.

Chunkfs [10] is a single-node file system with dynamic namespace partitioning, but with different goals and implementation. Chunkfs improves fault isolation and recovery by dividing the file system into multiple *chunks*, each containing one or more namespace subtrees that can be checked independently. Chunkfs uses *continuation inodes* to connect subtrees across chunks, similar to remote hardlinks in FlexGroups, except that they are restricted to a single node.

Like Chunkfs, SpanFS [16] stitches together multiple local file systems, called *domains*, into a single volume. SpanFS provides better MP scaling because locks and other resources are local to a single domain, allowing threads in different domains to execute without contention.

7 Conclusion

In this paper, we presented FlexGroup volumes, a distributed version of NetApp FlexVol volumes. FlexGroups achieve seamless scaling across the storage cluster even while simplifying the job of the storage administrator. FlexGroups leverage the maturity, stability, and feature richness of FlexVols. We described the core elements of the design: the infrastructure for remote hardlinks and the ingest heuristics that distribute newly created content. We evaluated FlexGroup performance using both benchmarks and archived customer usage data. The success of FlexGroups has been further validated by rapid customer adoption.

Acknowledgements: We thank the many WAFL engineers who contributed to these designs over the years; they are too many to list. We thank the anonymous reviewers and our the shepherd, Michael Factor, for their helpful comments and advice. We also thank Mike Montour and Robert Franz for their help with performance experiments, and we thank Jessie Wood for copy editing this paper.

References

[1] ANDERSON D.C., CHASE J.C., and VAHDAT A.M. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems*, *20*(1), pp. 25–48, February 2002.

[2] CORBETT P., ENGLISH B., GOEL A., GRCANAC T., KLEIMAN S., LEONG J., and SANKAR S. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–14. March 2004.

[3] CORBETT P.F. and FEITELSON D.G. The Vesta parallel file system. *ACM Transactions on Computer Systems*, *14*(3), pp. 225–264, August 1996.

[4] COREOS. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[5] CURTIS-MAURY M., DEVADAS V., FANG V., and KULKARNI A. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 419–434. November 2016.

[6] DOUCEUR J.R. and HOWELL J. Distributed directory service in the Farsite file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 321–334. November 2006.

[7] EDWARDS J.K., ELLARD D., EVERHART C., FAIR R., HAMILTON E., KAHN A., KANEVSKY A., LENTINI J., PRAKASH A., SMITH K.A., and ZAYAS E. FlexVol: Flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pp. 129–142. June 2008.

[8] EISLER M., CORBETT P., KAZAR M., and NYDICK D.S. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pp. 139–152. February 2007.

[9] GOEL A. and CORBETT P. RAID triple parity. *ACM SIGOPS Operating Systems Review*, *46*(3), pp. 41–49, 2012.

[10] HENSON V., VAN DE VEN A., GUD A., and BROWN Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the 2nd Conference on Hot Topics in System Dependency (HotDep)*. November 2006.

[11] HITZ D., LAU J., and MALCOLM M. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pp. 235–246. January 1994.

[12] How GlusterFS distribution works. https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/dht/.

[13] HOWARD J.H., KAZAR M.L., MENEES S.G., NICHOLS D.A., SATYANARAYANAN M., SIDEBOTHAM R.N., and WEST M.J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, *6*(1), pp. 51–81, February 1988.

[14] HPC IO Benchmark Repository. https://github.com/hpc/ior.

[15] HUNT P., KONAR M., JUNQUEIRA F.P., and REED B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pp. 145–158. June 2010.

[16] KANG J., BENLONG, WO T., YU W., DU L., MA S., and HUAE J. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pp. 249–261. July 2015.

[17] KESAVAN R., KUMAR H., and BHOWMIK S. WAFL Iron: Repairing live enterprise file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pp. 33–47. February 2018.

[18] KESAVAN R., SINGH R., GRUSECKI T., and PATEL Y. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–13. February 2017.

[19] KESAVAN R., SINGH R., GRUSECKI T., and PATEL Y. Efficient free space reclamation in WAFL. *ACM Transactions on Storage*, *13*(3), September 2017.

[20] KUMAR H., PATEL Y., KESAVAN R., and MAKAM S. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 197–211. February 2017.

[21] LAMPORT L. The part-time parliament. *ACM Transactions on Computer Systems*, *16*(2), pp. 133–169.

[22] LAMPSON B.W. and LOMET D. A new presumed commit optimization for two phase commit. In *Proceedings of the 8th International Conference on Very Large Data Bases (VLDB)*, pp. 630–640. August 1993.

[23] LANCASTER L. and ROWE A. Measuring real world data availability. In *Proceedings of the 15th Systems Administration Conference (LISA)*, pp. 93–100. December 2001.

[24] MCKUSICK M.K., NEVILLE-NEIL G.V., and WATSON R.N.M. *The Design and Implementation of the FreeBSD Operating System*, chapter 10. Addison Wesley, 2nd edition, 2015. ISBN 9780321968975.

[25] NETAPP, INC. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/data-ontap-8/, 2010.

[26] NETAPP, INC. Volume Move Express Guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2496251, May 2019.

[27] OUSTERHOUT J.K., CHERENSON A.R., DOUGLIS F., NELSON M.N., and WELCH B.B. The Sprite network operating system. *IEEE Computer*, *21*(2), pp. 23–36, February 1988.

[28] PATTERSON D.A., GIBSON G., and KATZ R.H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 SIGMOD International Conference on Management of Data*, pp. 109–116. June 1988.

[29] PATTERSON H., MANLEY S., FEDERWISCH M., HITZ D., KLEIMAN S., and OWARA S. SnapMirror: File system based asychronous mirroring for disaster recovery. *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pp. 117–129, January 2002.

[30] RODEH O., BACIK J., and MASON C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, *9*(3), 2013. ISSN 1553-3077. doi:10.1145/2501620.2501623.

[31] SANDBERG R., GOLDBERG D., KLEIMAN S., WALSH D., and LYON B. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer 1985 Technical Conference*, pp. 119–130. June 1985.

[32] SCHMUCK F. and HASKIN R. GPFS: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST)*, pp. 213–244. January 2002.

[33] SELTZER M.I., GANGER G.R., MCKUSICK M.K., SMITH K.A., SOULES C.A.N., and STEIN C.A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pp. 71–84. June 2000.

[34] SEVILLA M.A., WATKINS N., MALTZAHN C., NASSI I., BRANDT S.A., WEIL S.A., FARNUM G., and FINEBERG S. Mantle: A programmable metadata load balancer for the Ceph file system. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. November 2015.

[35] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014. https://www.spec.org/sfs2014/, 2017.

[36] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014 SP2 Run and Reporting Guide. https://www.spec.org/sfs2014/docs/runrules.pdf, 2017.

[37] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014 SP2 Users Guide. https://www.spec.org/sfs2014/docs/usersguide.pdf, 2017.

[38] THEKKATH C.A., MANN T., and LEE E.K. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 224–237. October 1997.

[39] WEIL S.A., BRANDT S.A., MILLER E.L., LONG D.D.E., and MALTZAHN C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 307–320. November 2006.

[40] WEIL S.A., BRANDT S.A., MILLER E.L., and MALTZAHN C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. November 2006.

[41] WELCH B. and OUSTERHOUT J. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pp. 184–189. May 1986.

[42] WELCH B.B., UNANGST M., ABBASI Z., GIBSON G.A., MUELLER B., SMALL J., ZELENKA J., and ZHOU B. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pp. 17–33. February 2008.

NETAPP, the NETAPP logo, and the marks listed at http://www.netapp.com/TM are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.

EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices

Xiang Gao[1], Mingkai Dong[2], Xie Miao[1], Wei Du[1], Chao Yu[1], and Haibo Chen[2,1]

[1]Huawei Technologies Co., Ltd.
[2]Shanghai Jiao Tong University

Abstract

Smartphones usually have limited storage and runtime memory. Compressed read-only file systems can dramatically decrease the storage used by read-only system resources. However, existing compressed read-only file systems use fixed-sized input compression, which causes significant I/O amplification and unnecessary computation. They also consume excessive runtime memory during decompression and deteriorate the performance when the runtime memory is scarce. In this paper, we describe EROFS[1], a new compression-friendly read-only file system that leverages fixed-sized output compression and memory-efficient decompression to achieve high performance with little extra memory overhead. We also report our experience of deploying EROFS on tens of millions of smartphones. Evaluation results show that EROFS outperforms existing compressed read-only file systems with various micro-benchmarks and reduces the boot time of real-world applications by up to 22.9% while nearly *halving* the storage usage.

1 Introduction

Low-end smartphones with relatively low price are still prevalent in the market [18, 25], especially in developing countries. At a price, such devices are usually equipped with limited resources in both capacity and performance. For example, a low-end Android smartphone may have 1-2GB runtime memory and 8-16GB slow eMMC storage [19, 21, 22]. Even worse, the Android operating system itself can consume more than 3GB in storage, leaving scarce storage space available to users [46]. Even for high-end smartphones, the increasing storage and runtime memory consumption of popular or resident apps usually render a device resource-scarce for both user-initiated and system-initiated operations.

File systems with compression support, or compressed file systems, can be used to release more space to the users by transparently compressing/decompressing file data upon accesses. However, such file systems usually consume more resources and yield notably worse performance during compression/decompression. Thus they are not suitable for resource-limited devices, especially smartphones, on which user experience has the top priority.

Fortunately, for partitions with read-only data, such as the */system*, */vendor* and */odm* partitions of Android, the file system can be made read-only to boost the performance by simplifying the structures and designs for file changes. However, existing compressed read-only file systems, such as Squashfs [10, 11], usually cause notable degradation on access performance and incur extra memory usage during decompression. One key issue is that such file systems use fixed-sized input compression, in which file data is divided into fixed-sized chunks (e.g., 128KB) and each chunk is compressed individually. The fixed-sized input compression incurs significant read amplification and excessively unnecessary computations (§2.2). Even worse, they usually require a huge amount of runtime memory, which is scarce on low-end smartphones or heavily-used high-end smartphones (§2.2).

To save the storage space and retain high performance with low memory overhead, we design and implement EROFS, an enhanced read-only file system with compression support. EROFS introduces the fixed-sized output compression, which compresses file data to multiple fixed-sized blocks, to significantly mitigate the read amplification problem and reduce unnecessary computations as much as possible. By exploiting the characteristics of compression algorithms (such as LZ4), EROFS designs different memory-efficient decompression schemes to reduce extra memory usage during the decompression. EROFS also adopts a set of optimizations that carefully ensure guaranteed user experience.

The main contributions of this paper include:

- A study of existing compressed file systems which reveals the performance issues on resource-hungry devices (§2).
- A fixed-sized output compression scheme that signifi-

[1]Short for Enhanced Read-Only File System. It has been upstreamed to Linux 4.19 as a major feature and integrated into Huawei's Smartphone Operating System (called EMUI) as a top feature (https://consumer.huawei.com/en/emui/) of version 9.1.

cantly mitigates the read amplification issue (§3.1).

- A set of novel decompression schemes for both memory-efficiency and high performance (§3.3).
- An evaluation of EROFS against other file systems to validate the effectiveness of EROFS (§5) and a study on the deployment experience of EROFS on tens of millions of smartphones (§6).

2 Background and Motivation

2.1 Low user-perceived storage space

Smartphones are usually resource-scarce due to the cost constraint. Meanwhile, the space occupied by the Android operating system is constantly increasing. Fig. 1 shows the */system* partition size in stock Android factory images [6] for different Android versions. The sparse image strips off all zero blocks and thus only contains all effective data; while the raw image is the actual space consumed once stored into the devices. From the figure, we can see the data size of the */system* partition increases from 184MB in Android 2.3.6 to 1.9GB in Android 9.0.0. Besides the trend of increasing the effective data size, we can also see a large number of zero blocks in Android 7 and 8, which also consume large space. For Android 9, the zero blocks are significantly less, which is due to the support of data block deduplication [20] in the ext4 file system. Besides the */system* partition shown in Fig. 1, there are other space-consuming partitions for Android such as */vendor*, */oem* and */odm* [8]. As reported in previous work [46], the space used by the whole Android system itself is increasing and far larger than what we show here. For example, Android 6.0.0 consumes 3.17GB storage after a factory-reset [46].

Meanwhile, the storage consumption of Android applications also keeps growing. As reported by Google Play, by early 2017, the average app size has quintupled compared with that at the time Google starts its Android application marketplace [45]. As a result, the storage capacity of low-end smartphones available for users is rather small. Further, many top apps for smartphones tend to consume a huge amount of memory, leaving only a small amount of memory for system-initiated operations even on a high-end smartphone.

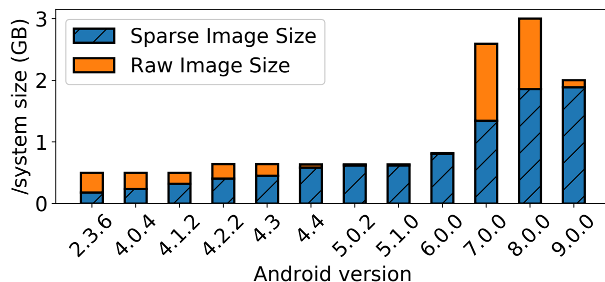


Fig. 1: Android /system partition sizes

Compressed file systems. One intuitive approach to unleashing more spaces for users is adopting compressed file systems, which exposes standard file interfaces to the applications but transparently compress and decompress file data during file writes and reads.

Btrfs [2] is a modern B-tree file system with compression support. When compression is enabled, the file data is divided into multiple 128KB chunks and compressed separately. Each of the compressed chunks will be stored in an extent, which is a run of contiguous blocks that store data sequentially. The locations of these extents are recorded as indexes in the B-tree structures. To read the file data, the corresponding extents are read from the storage and the whole chunks are decompressed. To update a file, the new data is compressed and written to new extents, and then the indexes are updated. To read the file data, Btrfs reads the corresponding extents from the storage and decompresses the whole chunks. To update a file, Btrfs compresses the new data, writes it to new extents, and updates the indexes.

Btrfs is a general-purpose file system, so its internal structures must consider efficient data modifications and cannot be aggressively optimized for compression. Furthermore, compression is not the only metric. The memory consumption during decompression should also be constrained.

For devices like smartphones, performance and responsiveness are important key metrics that cannot be compromised. Hence, with the burden of efficient data modification, Btrfs can hardly satisfy the requirements of both performance and compression efficiency, as we will show later in the evaluation (§5).

Compressed read-only file systems. Considering the access patterns of partitions in Android, we find that system resources are rarely modified once the Android operating system is installed. We can thus use compressed read-only file systems on read-only partitions to reduce the space consumption for system resources while retaining the performance. Unlike compressed read-write file systems which are complicated by data modifications, compressed read-only file systems exclude data updates by design, which exposes more opportunities for higher compression ratio and faster data reads.

Squashfs [11] is a widely-used compressed read-only file system in Linux with many features and moderate performance. It supports several compression algorithms, and the chunk (i.e., compression input) size can be chosen from 4KB to 1MB. In Squashfs, metadata can be compressed, and inodes and directories are stored more compactly. File data is compressed chunk by chunk, and the compressed data blocks are stored sequentially. The compressed sizes of each original data chunk are stored in a list within the inode. These sizes are used to locate the position of compressed blocks during decompression.

2.2 Deficiency of existing readonly file systems

Compressed read-only file systems are designed to minimize storage usage. However, applying existing compressed read-

only file systems on resource-scarce smartphones can induce significant overhead on both performance and memory consumption. For example, we first tried to use Squashfs for the read-only partitions on Android. While the system boots successfully with Squashfs, booting the camera application requires tens of seconds even with light background workloads.

Why is there such a huge performance slowdown? We conducted a detailed study of Squashfs with default configuration using microbenchmarks and uncover that the performance degradation mainly originates from two parts. The first one is I/O amplification. We used FIO [23] to evaluate the basic performance of Squashfs. When Android sequentially reads 16MB from the 1GB enwik9 [40] file stored in Squashfs, the actually issued I/O is 7.25MB. While the number looks decent regarding compression, Squashfs issues 165.27MB I/O reads when Android reads 16MB randomly. Moreover, when Android reads the first 4KB of every 128KB, reading 16M file data issued as much as 203.91MB I/O read. The difference suggests that when Squashfs reads some data that is not decompressed and cached before, the size of data requested is significantly amplified.

The second reason is extra memory consumption. The total memory consumption after sequentially reading the 1GB enwik9 file on Squashfs is about 1.35GB, which suggests that decompression in Squashfs requires a significant amount of temporary memory compared to the size of the original data needed. This causes high pressure to Android since memory is a key factor for user experience given that Android and its apps already consume a large amount of memory. On one hand, allocating memory during decompression may trigger memory swapping, which involves victim selections and I/Os with high cost. On the other hand, consuming much extra memory during decompression affects other components or applications by dropping their cached data or swapping out useful memory pages.

We further analyzed the design and implementation of Squashfs and found the following two defects.

Fixed-sized input compression. Existing file systems compress original data in a fixed-sized chunk, generating variable-sized compressed data. As shown in Fig. 2(a), Squashfs takes a fixed-sized data (e.g., a 128KB chunk) as the input of a single invocation of the compression algorithm. The compression algorithm then generates the compressed data whose size depends on the content of the input data. The compressed data of one file is usually compacted in the original data order, to reduce wasted space in the first and the last blocks of each compressed chunk.

Such a compression approach appears decent but has a notable deficiency due to amplified I/O and wasted computation. For example, in Fig. 2(b), the application wants to get the first byte of the 128KB chunk. To satisfy the application's request, the Squashfs has to read all compressed data from block 1 to block 7. Considering the minimal requested block size of the underlying storage devices is 4KB, the I/O is amplified 7 times! This is because the file system must read *all* related compressed blocks, even if the number of compressed blocks is very large. Even worse, even if not all data stored in the first block and the last block are useful for the decompression, they must be read from the storage altogether. In the example, the shadowed parts of block 1 and block 7 in Fig. 2(b) contribute nothing to the decompression but have to be read from the storage. Besides, the decompression process for useless data also causes huge CPU wastes that lead to high performance interference of other running apps (such as the Camera mentioned before).

One possible mitigation would be reducing the input chunk size to 4KB in Squashfs. While this might alleviate the I/O amplification, this non-trivially reduces the compression ratio and incurs higher CPU utilization, as we will show in section 5.

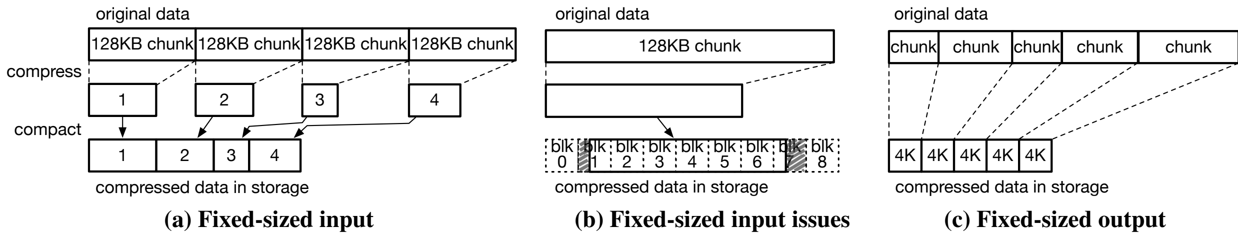
Massive memory consumption and data movements. The other defect we found is that Squashfs requires massive temporary memory during the decompression. Upon file read requests, Squashfs will first look up the metadata to get the number of related compressed blocks. It then allocates memory (e.g., the *buffer_head* structure) for each of the compressed blocks, and issues I/O reads to fetch the compressed blocks from the storage to the allocated *buffer_head*s. Since the buffers in *buffer_head*s of adjacent compressed blocks might not have continuous virtual addresses, Squashfs has to copy data in the *buffer_head*s of all compressed blocks to a single continuous buffer. Then, the compression algorithm decompresses all original data and puts them in a temporary output buffer. Finally, Squashfs copies the original data from the temporary output buffer to the corresponding page cache pages.

From the above routine, two pre-allocated temporary buffers are used and an array of *buffer_head*s are dynamically allocated for the decompression. The number of *buffer_head* needs to be large enough to store all compressed blocks. However, allocating such a large amount of memory can cause severe performance degradation under a low-memory situation.

In addition to extra memory allocation, there are two data movements during decompression: from the *buffer_head*s to the temporary input buffer, and from the temporary output buffer to the page cache. These two data movements also cause performance overhead since, most of the time, the compression/decompression algorithm is bottlenecked by memory accesses.

The above two defects in Squashfs reveal two challenges when designing a compressed read-only file system for resource-scarce smartphones.

- How to reduce I/O amplification during the decompression without sacrificing the compression ratio?
- How to reduce memory consumption during the decompression to prevent performance degradation?



(a) Fixed-sized input **(b) Fixed-sized input issues** **(c) Fixed-sized output**

Fig. 2: Compression approaches

3 EROFS:Enhanced Compressed File System

This section presents the design of EROFS, a compression-friendly readonly file system which overcomes the deficiency of prior systems. The key design of EROFS includes fixed-sized output compression, cached I/O and in-place I/O, and memory-efficient decompression.

3.1 Fixed-sized output compression

To overcome the read amplification incurred by the fixed-sized input compression, EROFS adopts a different compression approach: fixed-sized output compression.

To generate fixed-sized output, EROFS compresses the file data using a sliding window, whose size is a fixed value and can be configured during image generation. The compression algorithm is invoked multiple times until all file data is compressed. For example, with a 1MB sliding window, EROFS feeds the compression algorithm with 1MB original data at a time. The algorithm then compresses the original data as much as possible until all 1MB data is consumed or the consumed data can generate exactly 4KB compressed data. The remaining original data is combined with more data, forming another 1MB original data for the next invocation of compression. Fig. 2(c) depicts the fixed-sized output compression, in which variable-sized original data is compressed to 4KB blocks.

There are several benefits of using fixed-sized output compression compared to the fixed-sized input one. First, as what we will show in the evaluation (§5.3), it has better compression ratio under the same compression unit size. This is reasonable since the fixed-sized output compression can compress data as much as possible until the output limit is reached, while the fixed-sized input compression can only compress a fixed size of data at a time. Second, during the decompression, only the compressed blocks that contain the requested data will be read and processed. In the previous example where a single original block is requested, at most two compressed blocks will be read and decompressed. Third, as we will show later in §3.3, the fixed-sized output compression makes it possible to do in-place decompression, which further reduces the memory consumption in EROFS.

3.2 Cached I/O and in-place I/O

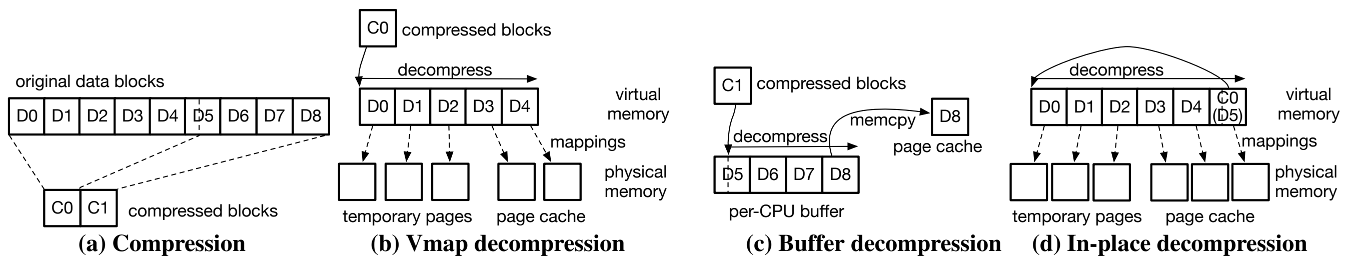
Before the actual decompression, EROFS needs space to store the compressed data retrieved from the storage. While this is costly for fixed-sized input compression due to excessive memory allocation and even page swapping, fixed-sized output compression would incur much less cost since EROFS clearly knows that each compression only retrieves up to two compressed blocks. There are two strategies for EROFS: cached I/O and in-place I/O. EROFS uses cached I/O for compressed blocks that will be partially decompressed. EROFS manages a special inode whose page cache stores compressed blocks indexed by the physical block number. Thus, for cached I/O, EROFS will allocate a page in the special inode's page cache to initiate the I/O request, so that the compressed data will be directly fetched to the allocated page by the storage driver.

For compressed blocks that will be completely decompressed, EROFS uses in-place I/O. On each file read, VFS will allocate pages in the page cache for the file system to put file data. For any one of these pages that contains no meaningful data before the decompression, we call it a *reusable page*. For in-place I/O, EROFS uses the last *reusable page* to initialize the I/O request.

Both I/O strategies are necessary. For cached I/O, partially decompressed blocks are cached in the special page cache, so that subsequent reads to the uncompressed part can use these blocks without invoking additional I/O requests. For blocks that are fully decompressed, they are unlikely to be used later since all decompressed data is stored in the page cache, which can serve subsequent reads without decompression. Thus, cached I/O vainly increases the memory spike due to page allocations for fully compressed blocks, while not contributing to subsequent file reads. In such cases, in-place I/O avoids unnecessary memory allocation, which relieves the memory pressure especially when there are many in-flight file read requests on different compressed blocks. Note that although it is possible to put the compressed block on the stack, it is not recommended to do so since the stack size is limited to be 16KB [14] and it is not easy to know how many bytes of the stack are still available.

3.3 Decompression

After loading compressed data into memory, we illustrate how EROFS decompresses data both fast and memory-efficiently. Examples in this section are based on Fig. 3(a) where the first five blocks (D0 to D4) and part of the block D5 are compressed to block C0, and the rest blocks are compressed to block C1. In this subsection, we only introduce



(a) Compression **(b) Vmap decompression** **(c) Buffer decompression** **(d) In-place decompression**

Fig. 3: Decompression

how a single compressed block is decompressed since, for read requests containing data in multiple compressed blocks, the compressed blocks are decompressed one by one similarly. For example, to read blocks D4 to D6 in Fig. 3(a), C0 is firstly decompressed to get D4 and the first part of D5; then C1 is decompressed to get the rest of D5 and D6.

Vmap decompression To get the data in block D3 and D4, EROFS first reads the compressed block C0 from the storage and stores it in the memory. Then EROFS will decompress it in following steps.

1. Find the largest needed block number that is stored in the compressed block (C0), which is the fifth block (D4) in the example. As an advantage, EROFS only needs to decompress the first five blocks (D0 to D4), rather than decompressing all original data blocks.
2. For each of the data blocks that need to be decompressed, find memory space to store them. In the example shown in Fig. 3(b), EROFS allocates three temporary physical pages to store D0, D1, and D2. For the requested two blocks, D3 and D4, EROFS reuses the two physical pages that have been allocated by VFS in the page cache.
3. Since the decompression algorithm requires continuous memory as the destination of decompression, EROFS maps physical pages prepared in the previous step into a continuous virtual memory area via the `vmap` interface.
4. If it's in-place I/O, in which case the compressed block (C0) is stored in the page cache page, EROFS also needs to copy the compressed data (C0) to a temporary per-CPU page so that the decompressed data won't overwrite the compressed data during the decompression.
5. Finally, the decompression algorithm is invoked, and data in the compressed block is extracted to the continuous memory area. After the compression, the three temporal physical pages and the virtual memory area can be reclaimed, and the requested data has already been written to the corresponding page cache pages.

Per-CPU buffer decompression The above decompression approach causes two problems. The first one is that it is still required to dynamically allocate physical memory pages, which increases the memory pressure on memory-constrained devices. The second problem is that using `vmap` and `vunmap` on each decompression is inefficient.

EROFS leverages per-CPU buffers to mitigate the problems when the decompressed data is less the four pages. As shown in Fig. 3(c), a four-page memory buffer is preallocated for each CPU as the per-CPU buffer. For decompression that extracts no more than four blocks of data, EROFS decompresses the data to the per-CPU buffer and then copy the requested data to the page cache pages. In the example demonstrated in Fig. 3(c), data in block D8 is requested. The compressed data in C1 is directly decompressed to the per-CPU buffer, and the content of D8 is copied to the page cache page via `memcpy`.

The length of the per-CPU buffer is empirically decided, but it can effectively eliminate memory allocations since the per-CPU buffer can be reused across different decompressions. The per-CPU buffer decompression is a cost-effective trade-off which mitigates issues in the vmap decompression while introducing extra memory copies.

Rolling decompression To avoid the overhead of `vmap` and `vunmap` and eliminate other dynamic page allocations, EROFS allocates a large virtual memory area[2] and 16 physical pages for each CPU.

Before each compression, EROFS uses the 16 physical pages, along with the physical pages of the page cache to fill in the VM area, so that step 2 and step 3 of the vmap decompression can be skipped.

EROFS uses LZ4 as the compression algorithm, which needs to look backward at no more than 64KB of the decompressed data [7]. Thus, for a compression that extracts more than 16 pages, EROFS can reuse the physical page mapped 16 virtual pages (i.e., 64KB) before. For example, in Fig. 4, the virtual addresses to store blocks D0 to D15 are backed by the 16 physical pages. The virtual page of D16 can be backed by the same physical page with D0 since each virtual address in D16 is 64KB away from the corresponding address in D0. D17 is backed in the same way by the physical page used by D1. D18, which is requested by the file read, uses the physical page of the page cache.

As a result, 16 physical pages are sufficient for any decompression cases by using such a rolling decompression.

[2] A virtual memory of 256 pages is sufficient for all the workloads we have met.

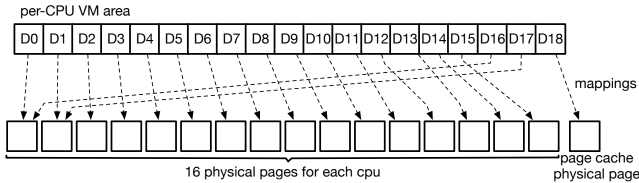


Fig. 4: An example of rolling decompression

In-place decompression In step 4 of the vmap decompression approach, the compressed data is moved to a temporary per-CPU page to avoid data not yet compressed from being overwritten by the compressed data (Fig. 5).

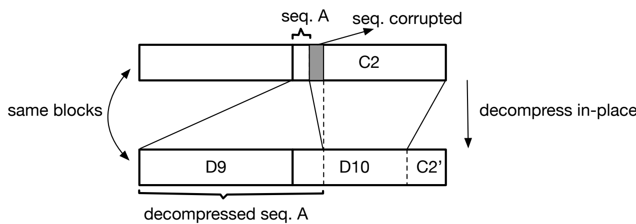


Fig. 5: An example compressed block (C0) that cannot be decompressed in-place. The decompressed data of sequence A corrupts the next sequence (the shadow part) which has not been decompressed.

However, if such a situation will never happen for a compressed block, we can decompress it in-place to avoid the extra memory allocation and memory copies. EROFS simulates the decompression during *mkfs* and marks whether a compressed block can be decompressed in-place in the block index. During the decompression of a block that can be decompressed in-place, step 4 is skipped. In our tested workload enwik9 in §5, 99.6% compressed blocks can be decompressed in-place; thus, most blocks can benefit from the in-place decompression as long as they are retrieved by in-place I/O.

4 Implementation

We have implemented EROFS as a Linux file system and upstreamed the common part of EROFS to Linux kernel[3].

In the current implementation, we use 4KB as the fixed output size since it is the minimal unit of page management and storage data transfers, and thus I/O amplification can be minimized. We support LZ4 (v1.8.3) as the compression algorithm since it has the fastest decompression speed and good compression ratio in our case. Other compression algorithms, such as LZO, can be supported once they are modified to provide fixed-sized output compression interfaces. Only the file data is compressed in EROFS; metadata such as inode and directory entries is stored without compression.

Currently, EROFS is still under active development and new features are constantly shipped into the smartphones after a rigorous commercial testing process. Hence, we introduce two versions of EROFS: 1) the latest version with all features and optimizations presented in this paper; 2) the commercially-deployed version, which has all features and optimizations except the rolling decompression and the in-place decompression. The two versions are also different in decompression policies which we will illustrate in §4.2.

4.1 EROFS image layout

Fig. 6 shows the layout of an EROFS image. As in other file systems, a super block is located at the beginning of the image. Following the super block, metadata and data may be stored in a mixed style without constraints on the order.

In the current implementation, metadata and data of a file are stored together for better locality. For each file, as shown in Fig. 6, an inode is stored at the beginning, followed by blocks containing the extended attributes (i.e., xattrs) and the block index. Blocks for compressed or uncompressed file data (encoded blocks) are stored at the end of each file.

Since an inode can be placed anywhere in the image, the inode number is calculated from the position of an inode, so that the inode can be quickly located. Blocks for xattrs and the block index are omitted if a file contains no xattrs or is uncompressed. Further, xattrs, the block index and file data can also be inlined within an inode if possible, which reduces storage overhead and decreases the number of I/O requests since the inlined data/metadata is read along with the inode.

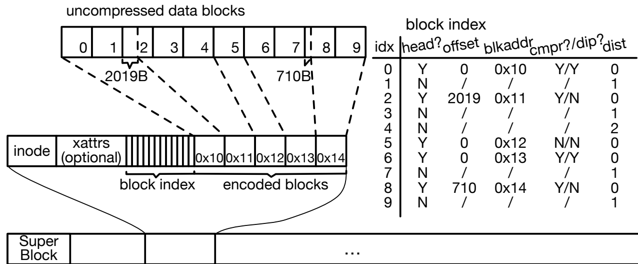
The block index is used to quickly locate the corresponding encoded block for read requests. Fig. 6 shows an example block index for a regular file containing ten blocks before compression. The block index is an array of 8B-length entries, each of which corresponds to a data block before compression. Each entry indicates whether the corresponding data block is the head block (the boolean `head` field in Fig. 6), which starts a new encoded block. If so, the encoded block address (`blkaddr`), the offset of the first byte in the new encoded block (`offset`), whether the encoded block is compressed (`cmpr`), and whether the block can be decompressed in-place (`dip`) are also stored. If not, there must be a head block before the uncompressed block, and the block number difference to the head block is recorded in `dist`.

For a read request to an uncompressed data block, EROFS gets the block index entry according to the requested block number. For a head block, EROFS decompresses data from the block at `blkaddr`, and if the `offset` is non-zero, EROFS may also need to decompress from the nearest encode block stored before the `blkaddr`. For a non-head block, EROFS calculates the location of the corresponding head block according to the stored `dist`, and starts to decompress until the requested block data is decompressed.

Some data blocks (e.g., block 5 in Fig. 6), which are larger after compression, are not compressed and directly stored as encoded blocks. For these cases, the corresponding `cmpr` fields are set to false (i.e., "N" in the figure).

[3]Since some optimizations are not yet upstreamed, we also make the latest version of the code available at https://github.com/erofs/atc19-erofs and https://github.com/erofs/atc19-mkfs.

Directories are stored similarly as the regular files, except that there is no block index, and the encoded blocks are used to store uncompressed directory entries. For better locality of random accesses in directories, EROFS puts all dirent headers (e.g., inode number, file type, and name length) at the beginning of directory entries part, and places filenames after those headers.



idx	head?	offset	blkaddr	cmpr?/dip?	dist
0	Y	0	0x10	Y/Y	0
1	N	/	/	/	1
2	Y	2019	0x11	Y/N	0
3	N	/	/	/	1
4	N	/	/	/	2
5	Y	0	0x12	N/N	0
6	Y	0	0x13	Y/Y	0
7	N	/	/	/	1
8	Y	710	0x14	Y/N	0
9	N	/	/	/	1

Fig. 6: EROFS image layout and the block index

4.2 Decompression policy

Two versions of EROFS have different decompression policies. The commercially-deployed EROFS uses the per-CPU buffer decompression if less than four original data blocks are to be extracted; otherwise, the vmap decompression is used.

In the latest EROFS, all four decompression approaches are implemented. If there is no more than one data block to be extracted, the per-CPU buffer decompression is chosen. Otherwise, if a compressed block is retrieved using in-place I/O and can be decompressed in-place, EROFS employs the in-place decompression approach which avoids unnecessary memory allocations and memory copies. For other cases where the decompressed blocks can fit in the pre-allocated VM area, EROFS uses the rolling decompression since it beats the vmap decompression with less memory allocation overheads. For any other cases, the vmap decompression approach is adopted.

4.3 Optimizations

Index memory optimization It is possible that EROFS compresses hundreds of pages of original data into a single compressed block. In such a case, EROFS needs hundreds of pointers to keep track of where each page of the original data should be stored. These pointers can consume a large amount of memory. To address such a problem, EROFS tries to store the information with the help of *reusable pages*. If there are more than one VFS allocated pages are *reusable*, EROFS uses the last page to store the compressed data, and the other pages to store some of these pointers during the I/O. Before the actual decompression, these pointers are moved onto the stack, so that the *reusable pages* are free to store the decompressed data.

Scheduling optimization Decompression requires a relatively long time. Thus it is not suitable to be done within the interrupt context. In some file systems, such as Btrfs [2], when compressed data has been fetched to memory, a dedicated thread will be woken up to decompress the data. When the decompression is finished, the reader thread which issues the I/O will be woken up to get the decompressed data from the page cache. To reduce scheduling overhead, EROFS decompresses data in the reader thread, without dedicated threads for decompression. Thus once the compressed data has been fetched to memory, the reader thread will be directly woken up and start decompressing the data.

Cohort decompression Several requests can be in-flight simultaneously. If an original data block is requested on thread A and the corresponding compressed block is being decompressed by another thread named thread B, rather than decompressing the data by itself, thread A can wait for thread B to finish the decompression, and then directly read the decompressed data from the page cache. Such cooperation reuses the decompressed data and prevents a single data being decompressed multiple times.

Image patching Although EROFS is a compressed read-only file system, there are cases such as system upgrade or security patching where the data stored in EROFS needs to be updated. EROFS provides a feature called image patching, which supports partial data updates. Usually, modifying a single bit in the original file data might cause a huge amount of scattered modifications in the compressed data. Instead of modification in-place, image patching places updated data at the end of the EROFS image, and when the corresponding file data blocks are requested, the origin data blocks are firstly decompressed and then the updated data is applied to overwrite the decompressed data in memory. In this way, image patching prevents scattering of changes and supports partial data updates without re-compressing the while file system.

5 Evaluation

We have conducted a set of experiments to answer the following questions:

- How does compression affect the performance of file system read accesses?
- How much memory does EROFS consume during decompression?
- How does EROFS affect the boot time on real-world applications?

5.1 Evaluation setup

By default, we conduct experiments on an ARM development board, HiKey 960, running Android 9 Pie with Linux kernel 4.14. The board is equipped with Kirin 960 (four Cortex-A73 big cores and four Cortex-A53 little cores), 3GB Hynix LPDDR4 memory and 32GB Samsung UFS storage. We also evaluate on two kinds of smartphones in some experiments. The low-end smartphones are equipped with MT6765 (eight Cortex-A53 cores), 2GB memory and 32GB

eMMC storage. High-end smartphones run with Kirin 980 (four Cortex-A76 cores and four Cortex-A55 cores), 6GB memory and 64GB UFS storage.

For micro-benchmarks, we run FIO [23], a flexible I/O tester, on various file systems including EROFS, Squashfs, Btrfs, Ext4, and F2FS. We use the latest version of EROFS for micro-benchmark evaluation. Among these file systems, EROFS and Squashfs are designed to be compressed read-only file systems; Btrfs is a file system with compression support, but it is not a file system designed for read-only data; Ext4 is the default file system used by Android; F2FS is a file system designed for mobiles and is widely used in some smartphones.

EROFS is configured to use 4KB-sized output compression with LZ4. Squashfs is configured to use LZ4 with 4KB, 8KB, 16KB, and 128KB compression chunk sizes, indicated by Squashfs-4K, Squashfs-8K, Squashfs-16K, and Squashfs-128K, respectively. Btrfs is configured to run in read-only mode without data integrity checks for a fair comparison. The compression algorithm used by Btrfs is LZO, since Btrfs does not support LZ4. Both Ext4 and F2FS are used without compression in the experiments since they do not support it.

For real-world applications, we compare EROFS with Ext4, since Ext4 is now the default file system used by Android [17]. We use the commercial version for real-world evaluation since it takes time to ship the latest version to smartphones. We also tried to use Squashfs on Android. However, it costed too much CPU and memory resources, and when trying to run a camera application, the phone froze for tens of seconds before it finally failed.

5.2 Micro-benchmarks

We use FIO to show the basic I/O efficiency of different file systems. In this experiment, we use enwik9 [40] as the workload, which is the first 10^9 bytes of the English Wikipedia dump. We store the file in different file systems and read the file to test the file system read throughput. Each read is a 4KB buffered read. We test the throughput under three scenarios: sequential read, random read, and stride read. For the sequential read, we read the file 4KB by 4KB sequentially; thus the following reads are highly likely to hit in cache since the data is already loaded in the memory by previous decompression or prefetching (i.e., readahead). For the random read, we randomly read the whole file; thus the reads can hit in the cache if the data is already decompressed by previous reads. The last scenario is the stride read, in which we only read the first 4KB in every 128KB data. Since the largest compression chunk is 128KB, stride reads will not hit in cache[4]. We test stride reads to illustrate the worst-case performance for compressed file systems.

Before each test, the page cache is dropped to reduce interference. All tests are done at least ten times, and the average throughputs are reported. The max relative standard deviation is 17.3% for stride reads on A53 cores and 5.1% for the rest results. Fig. 7 shows the following results we observed.

Btrfs performs worst in all tests compared with EROFS and Squashfs-128K, since it is designed neither for compression nor for read-only data. On one hand, Btrfs does not take advantage of the read-only property and has to consider updates; thus it is outperformed by the compressed read-only file systems EROFS and Squashfs-128K. On the other hand, decompression in Btrfs incurs notable performance overhead compared to Ext4 and F2FS which do not need to decompress data during reads. This is reasonable since Btrfs is not designed to be a compressed read-only file system.

Btrfs performs better than other configurations of Squashfs for sequential reads, which is caused by its larger compression chunk (128KB). The advantages shrink in random reads where prefetching does not work; the advantage disappears in stride reads where decompressing more data than requested becomes the burden.

Overall, this result shows the inefficiency of using general file systems with compression support for read-only data and emphasizes the necessity of designing compressed read-only file systems.

As the size of compression input increases, the performance of Squashfs increases for random reads and sequential reads, but decreases for stride reads. The main reason for this phenomenon is the locality and cache. Since file systems have enough memory to cache file data in this experiment, all decompressed data will be cached and possibly be read in the future. Thus for random reads and sequential reads, the larger-sized data is decompressed, more future reads will hit the cache. That is basically the reason why the Squashfs throughputs grow as the compression chunk size increases.

Since both sequential and random reads will read the whole file, there is only a little performance difference, which is caused by the good locality and prefetching.

For stride reads, however, FIO only reads the first 4KB data for each 128KB data, which eradicates the benefits of memory cache since all the data decompressed but not requested will never be used in the future. Thus the more irrelevant data is read and decompressed, the more time and resource are wasted, yielding worse performance. That explains why the throughput drops with the increase of the compression chunk size for Squashfs.

EROFS performs best in most of the tests among file systems with compression support and sometimes outperforms file systems that do not compress data. For sequential reads, EROFS exhibits the best performance among compressed file systems. Most wins come from the design of fixed-sized output compression and the elimination of unnecessary memory allocations and data movements compared with Squashfs. For random reads, EROFS is outperformed by Squashfs-128K since the latter can decompress and cache the whole file during the test, while EROFS only benefits from cached

[4] In enwik9 and silesia.tar, no more than 128KB data is compressed to a single block in EROFS.

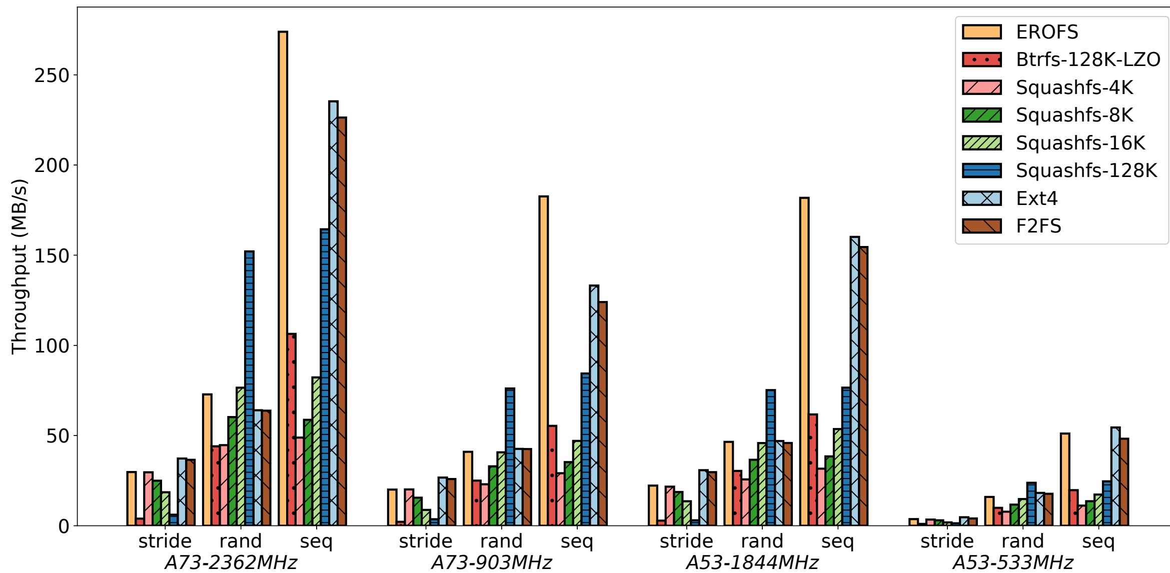


Fig. 7: FIO micro-benchmark results under three read patterns at four CPU frequencies

I/Os. However, EROFS still performs better than other compressed file systems. For stride reads, since the prefetching is barely useful, EROFS still yields the best throughput among compressed file systems, but the win is limited.

Compared with Ext4 and F2FS without compression support, EROFS always performs comparably with and even outperforms them (e.g., the sequential reads on A73 cores). The reason is that even if EROFS needs to decompress data, it reads much less data from the storage thanks to compression.

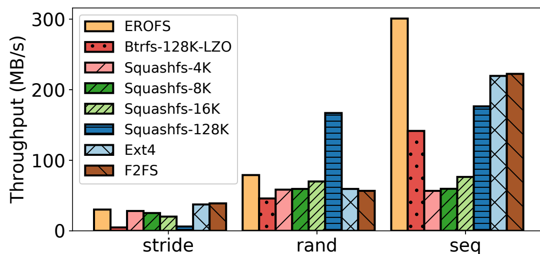
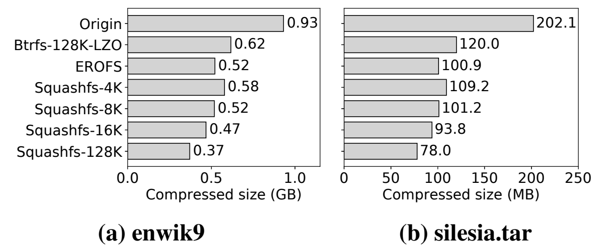


Fig. 8: FIO micro-benchmark results for silesia.tar

We also use the workload, *silesia.tar* [9], to conduct the same experiment. Silesia.tar is a tarball of the silesia compression corpus, which covers typical data types used nowadays. The results show the same trends as enwik9, so we only present the result for A73 cores at 2362MHz in Fig. 8.

5.3 Compression ratio and memory usage

We also evaluated the compression ratio and memory consumption during the decompression of each file system. We use both enwik9 and silesia.tar to present the compression ratio of different file systems. Fig. 9(a) and Fig. 9(b) show the number of bytes used on each file system for enwik9 and silesia.tar. The *origin* line in both figures represents the size of the uncompressed workload file, which is 953.67MB for enwik9 and 202.1MB for silesia.tar. Currently, EROFS only supports 4KB-sized output compression, but compared with Squashfs-4K, the compressed size is 10% and 9% smaller for the two workloads. The figure also matches the facts that the larger the compression unit, the better the compression ratio.



(a) enwik9 **(b) silesia.tar**

Fig. 9: Compressed size

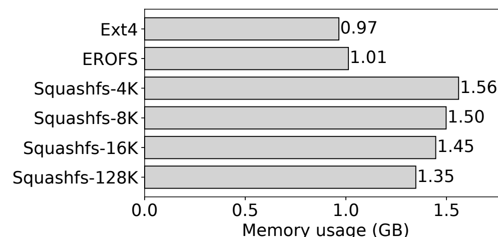


Fig. 10: Decompression memory usage

Fig. 10 shows the memory used after decompressing the enwik9 file. The test is conducted as follows: boot the machine, mount the file system, read the file stored in the file system, check the memory used, and then reboot.

Since the file is roughly 1GB, the remainders are either used by other parts of the operating system or temporarily

Table 1: I/O amount under different read patterns

I/O (MB)	seq-read	rand-read	stride-read
Requested	16.00	16.00	16.00
Squashfs-4K	10.65	26.19	26.23
Squashfs-8K	9.82	33.52	34.08
Squashfs-16K	9.05	46.42	48.32
Squashfs-128K	7.25	165.27	203.91
EROFS	10.14	26.12	25.93

Table 2: I/O patterns

I/O size	=4K	<=8K	<=16K	<128K	=128K	>128K
%	19.0	23.9	30.4	78.9	19.9	1.2

used by the file system. Besides EROFS and Squashfs, we also tested Ext4 as the baseline. From the figure, we can see that compared to Ext4, the memory overhead for four configurations of Squashfs ranges from 39.6% to 61.6%. However, memory used by EROFS is only slightly higher than the Ext4 (about 4.9%). The result shows that EROFS has much lower memory spikes than Squashfs and proves the effectiveness of memory-friendly decompression of EROFS.

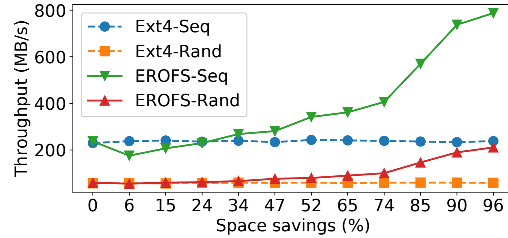
In the test, there is only one file to be decompressed, and we allocate abundant memory to ensure that no memory reclamation or swapping will happen during the decompression. However, in a real-world scenario where more files will be decompressed simultaneously, more memory will be needed by the decompression of Squashfs. Once the available memory is scarce, memory reclamation or swapping may happen, which is very expansive and affects not only the file systems, but also other components or applications in the whole system. Thus in real-world scenarios, the advantages of EROFS, which uses as little as memory during the decompression, will be more remarkable.

5.4 I/O amplification and I/O patterns

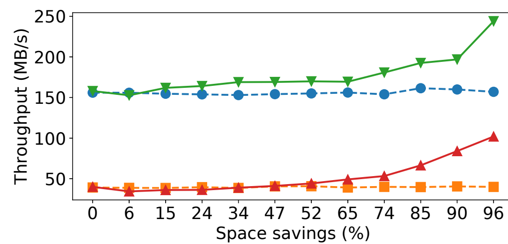
We reran tests mentioned in §2.2 on EROFS and different Squashfs configurations. Table 1 lists the actual I/O issued when reading 16MB file data under three read patterns. EROFS issued the least I/O for random reads and stride reads. Yet, since Squashfs-8K, Squashfs-16K and Squashfs-128K have a better compression ratio, they read less data than EROFS for sequential reads. In summary, EROFS reduces the I/O amplification for most cases compared with Squashfs, especially for random reads and stride reads.

We further identified the I/O pattern in a simulated real-world environment to illustrate how I/O amplification will affect real-world applications. We installed 100 apps and ran the Monkey tool [13] to randomly tap the screen once per second for 3 hours. We collected the I/O sizes passed to the `readpage` and `readpages` interfaces and show the proportion of different I/O sizes in Table 2. The result shows that there are quite a lot of I/Os (30.4%) with sizes no more than 16K, which we consider as random I/Os. The amount of random I/Os is reasonable since as the system keeps running for a long time, some pages in the applications' page cache are reclaimed due to memory shortage. The insignificant amount of random I/Os emphasizes the importance of EROFS's effort of reducing the I/O amplification.

5.5 Throughput and space savings



(a) Throughput on A73 cores 2362MHz



(b) Throughput on A73 cores 903MHz

Fig. 11: Throughput under different space savings

Fig. 11 depicts the throughput of EROFS and Ext4 under different space savings. The space savings is the value of space reduced by the compression divided by the size of original data; thus, a larger space savings indicates more space saved. For simplicity, we only show the results for big cores, since similar trends are presented on little cores.

For the test, we collected blocks in our compressed partitions and check their space savings. When we found a block that matches our expected space savings, the original data was decompressed and duplicated multiple times to form a roughly 512MB file. Then we stored the file in EROFS and read it to get the read throughput under its space savings. We also stored the file in Ext4 and got the corresponding read throughput for comparison.

Generally, the throughput of Ext4 remains stable during the test and the performance of EROFS increases together with space savings. EROFS achieves much better throughput than Ext4 when the space savings is high enough. In such cases, a single compressed block can be decompressed to dozens of blocks. Thus, the number of I/O requests is notably reduced, leading to higher performance. While when the space savings is low, the performance of EROFS is similar to Ext4 for random reads and worse than Ext4 for sequential reads. This is the result of the conjunction of I/O costs and decompression computation costs. For random reads, the I/O is more expensive than the decompression computation. While for sequential reads, due to prefetching, I/O is less costly and the computation cost dominates when the space

savings is low.

5.6 Different decompression approaches and optimization

To illustrate the effect of different decompression approaches, we ran FIO sequential reads on the Kirin 980 smartphone with A76 cores at 2600MHz. The vmap decompression approach serves file reads at 726.5MB/s while the per-CPU buffer decompression yields throughput of 736.5MB/s. File data is read at 769.7MB/s in the latest EROFS with the rolling decompression and the in-place decompression added.

We also evaluated the effect of scheduling optimization in §4.3 with the same configuration. In the random read workload, the average throughput of EROFS without the scheduling optimization is 64.49MB/s, while with the optimization, the performance improves 9.5% to be 70.61MB/s.

5.7 Real-world applications

For real-world applications, we ran modified Android 9 Pie on both low-end smartphones and high-end smartphones, whose hardware configurations are listed in §5.1. Android system partitions such as */system*, */vendor*, and */odm* are compressed with EROFS, and the space savings ranges from 30% to 35%. We tested the boot time of thirteen popular applications required by the production team. We compared application boot time on EROFS to those on Ext4 and present relative boot time in Table 3. On average, EROFS reduces the boot time by 5.0% for low-end smartphones and 2.3% for high-end smartphones compared with Ext4.

We also conducted the same test while running FIO as the background workload to simulate real-world scenarios. In the FIO workloads, four threads randomly read and write individual files with rate limited at 256KB/s for both reads and writes. The last two rows in Table 3 show the boot time with FIO workloads, where the reduction of boot time is 3.2% and 10.9% for low-end and high-end smartphones, respectively.

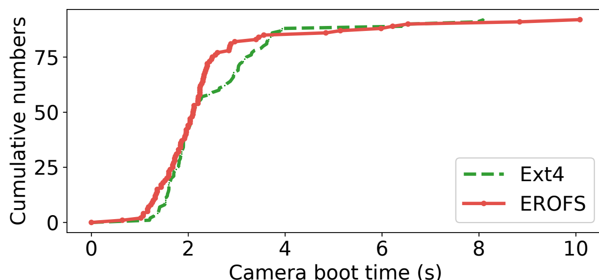


Fig. 12: Camera boot time

Other than the boot time of various applications, we also tested the boot time distribution of the camera application on the aforementioned high-end smartphones. To simulate the situation where memory is scarce, we ran a program in the background which continuously allocates memory and fills in garbage data. We waited until the program consumed all zram in the system before starting the experiment, and kept it running during the evaluation. In the experiment, we booted several applications in turn and recorded the boot time when it's the camera's turn to boot. We collected each time of 92 camera boots for both EROFS and Ext4, and present the cumulative distribution in Fig. 12. The camera application running on EROFS boots faster than on Ext4 for more than 90% of the boots, while the longest boot time on EROFS is worse than that on Ext4. We think the result is acceptable since EROFS saves the storage space while reduces the boot time in most of the cases.

6 Experience over deployment

EROFS has been deployed to tens of millions of smartphones. Here, we report some experiences during the deployment.

Optimizing for all cases, not only common cases. Deploying a new file system to replace an existing one is much harder than we first imagine. The reason is that a commercial product like a smartphone needs to retain the benefit and features of an existing file system. Hence, we need to carefully optimize EROFS for all cases instead of common cases to avoid performance degradation even in some rare cases.

For example, the performance of reading some files on EROFS is slightly worse than on Ext4. To optimize, we leave files with low compression ratio uncompressed for better performance. Further, we collect access frequencies of file blocks from anonymous beta users and store them in dedicated files. According to the frequency information, we pre-decompress the most frequently requested parts of compressed files and pin them in the memory to balance the storage consumption and the performance. As a result, the performance of EROFS can be as good as, and sometimes better than Ext4, while the storage consumption is significantly reduced.

Incomplete implementation leads to performance abnormalities and failures in real-world scenarios. We tested EROFS after the main functionalities have been implemented. However, several kinds of malfunctions happened during real-world tests.

One example is that after the smartphone runs on EROFS for days, several applications become extremely slow at times. Eventually, we found that the root cause is the missing implementation of page migration in EROFS. Page migration is invoked by the memory management subsystem to ask file systems to move their data somewhere else. Most of the time, the page migration will not be triggered and thus leaving it unimplemented is benign. However, when the memory is fragmented, which is the case when the bug happens, the functionality of page migration is crucial to the success of allocating contiguous memory in the system. The issue was solved after we implemented the page migration in EROFS.

Bottlenecks shift on different platforms. We developed

Table 3: **Relative boot time of thirteen applications on low-end and high-end smartphones. Each number in the table is the average value of at least five boots. Negative numbers show the boot time reduction (in %) compared with Ext4, while positive numbers indicate the boot time is prolonged (in %). FIO workloads are running in the background for cases with 'w/ FIO' suffix.**

App. #	1	2	3	4	5	6	7	8	9	10	11	12	13
Low-end	**-16.4**	-3.5	+4.2	-4.0	-7.5	-1.4	-6.8	+6.3	-2.2	**-18.4**	-3.3	-7.6	-4.5
High-end	-1.8	-0.7	-2.1	-1.8	**-12.3**	-3.7	+1.2	-8.0	-2.8	+0.7	+2.0	-2.7	+1.9
Low-end w/ FIO	-2.8	**-12.9**	-5.4	+3.9	-7.6	+3.7	+4.4	-2.6	+9.9	+4.0	**-11.1**	**-10.3**	**-15.1**
High-end w/ FIO	-4.6	**-14.1**	**-10.7**	**-19.3**	-7.0	**-11.0**	**-15.0**	+0.8	**-22.9**	-5.0	**-18.9**	-0.7	**-13.2**

early versions of EROFS on high-end smartphones where resources are abundant, and tuned it to use as fewer resources as possible. However, when we adopt the tuned EROFS to low-end smartphones, the performance is lower than we expected. This is surprising since we have already considered the limited resources and EROFS should work well. At last, the trouble-maker turned out to be the scheduler. Scheduling on low-end smartphones is much more costly and becomes the bottleneck of EROFS's decompression, which motivated us to introduce the scheduling optimization described in §4.3. Different platforms not only reflect resource limitation directly on the amount of memory available or how fast processors can run, but they can also shift the bottleneck of software.

7 Related Work

Other compressed file systems. Several other file systems support compression. AXFS [24] is a compressed read-only file system. It is designed to enable execute-in-place (XIP), which is not supported by common smartphone storage like eMMC or UFS. CramFS [3] is another compressed read-only file system, which is designed to be simple and space-efficient. However, it also has several limitations such as limited file size. Cramfs was once obsoleted by Squashfs in the Linux kernel [4] and then revived for XIP [5], which is not supported by common smartphone storage like eMMC or UFS. LeCramFS [27] extends CramFS for better read performance and memory efficiency on flash memory. However, the compression ratio is reduced, and LeCramFS generates much larger images.

JFFS2 [15], and UBIFS [12] are two file systems designed for flash memory. Although they support compression, they have to manage the wear-leveling, address translation, and garbage collection for NAND flash. Because all such features are already provided by the eMMC and UFS firmware, EROFS is much simpler and faster than JFFS2 and UBIFS.

Bcachefs [1] is a file system with an emphasis on reliability and robustness. ZFS [16] is a full-fledged file system designed by Sun Microsystems for Solaris. Although they both support compression, they have to consider updates on compressed files, which has a similar issue with Btrfs.

File system and storage for smartphones. File system and storage for smartphones have long been a hot topic. For example, Kim et al. [33] illustrated that storage can affect application performance on smartphones and proposed several approaches to mitigating the performance impact. Jeong et al. [31] uncovered and mitigated the journaling of journal (JOJ) anomaly by overhauling file systems on smartphones, which has generated several follow-up efforts [36, 38, 44]. They further identified Quasi-Asynchronous I/O in smartphone file systems and boosted them for responsiveness [28]. SmartIO [41] reduces the application delay by prioritizing reads over writes. MobiFS [43] is a memory-centric design for smartphone data storage that improves response time and energy consumption.

There has also been much work [29, 30, 34, 35, 37] providing benchmarking frameworks to evaluate file systems and storage on smartphones. Due to the emergence of non-volatile memory (NVM), recent researchers [26, 32, 39, 42, 47] also investigated how NVM can be used on smartphones.

8 Conclusion and Future Work

We introduce EROFS, a new compressed read-only file system designed for smartphones with limited resources. EROFS provides a comparable compression ratio while having much higher performance and less extra memory overhead compared to Squashfs. With fixed-sized output compression and fast and memory-efficient decompression, EROFS can store system code and resources with less storage usage and sometimes even better performance compared with file systems without compression support. Evaluation shows that apps on a system installed on EROFS can boot comparably or even faster compared with on Ext4. EROFS has been merged to the mainline Linux and has been deployed and used in tens of millions of smartphones. Currently, EROFS is still under active development, and we are continuously adding new features, such as deduplication, extended file statistics, fiemap, and EROFS-fuse in future versions of EROFS.

Acknowledgment

We thank our shepherd Ric Wheeler and the anonymous reviewers for the constructive comments, Guifu Li for helping prototype the `mkfs` utility, and Qiuyang Sun for his help with the early testing and evaluation. Haibo Chen is the corresponding author.

References

[1] Bcachefs. https://bcachefs.org.

[2] Btrfs: Main page. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[3] Cramfs - cram a filesystem onto a small ROM. https://www.kernel.org/doc/Documentation/filesystems/cramfs.txt.

[4] Cramfs: mark as obsolete. https://lkml.org/lkml/2013/9/4/79.

[5] Cramfs refresh for embedded usage. https://lkml.org/lkml/2017/8/11/726.

[6] Factory images for Nexus and Pixel devices. https://developers.google.com/android/images.

[7] LZ4 block format description. https://github.com/lz4/lz4/blob/master/doc/lz4_Block_format.md.

[8] Partitions and images. https://source.android.com/devices/bootloader/partitions-images.

[9] Silesia compression corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.

[10] SQUASHFS. http://squashfs.sourceforge.net.

[11] SQUASHFS 4.0 filesystem. https://www.kernel.org/doc/Documentation/filesystems/squashfs.txt.

[12] UBIFS file system. https://www.kernel.org/doc/Documentation/filesystems/ubifs.txt.

[13] UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey.

[14] x86_64: expand kernel stack to 16K. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6538b8ea886e472f4431db8ca1d60478f838d14b.

[15] JFFS2: The journalling flash file system, version 2. http://www.sourceware.org/jffs2/, 2003.

[16] ZFS: the last word in file systems. https://web.archive.org/web/20060428092023/http://www.sun.com/2004-0914/feature/, 2004.

[17] Saving data safely. https://android-developers.googleblog.com/2010/12/saving-data-safely.html, 2010.

[18] Android one was conceived with india in mind, says Google's Sundar Pichai. https://gadgets.ndtv.com/mobiles/news/googles-sundar-pichai-on-android-one-in-an-exclusive-chat-with-ndtvs-vikram-chandra-592062, 2014.

[19] HUAWEI Y3 2018. https://consumer.huawei.com/za/phones/y3-2018/specs/, 2018.

[20] Libext2fs: add EXT2_FLAG_SHARE_DUP to de-duplicate data blocks. https://android-review.googlesource.com/c/platform/external/e2fsprogs/+/642333, 2018.

[21] Nokia 2.1 - long lasting entertainment. https://www.nokia.com/phones/en_int/nokia-2, 2018.

[22] Samsung unveils the Galaxy J2 core; an introductory smartphone packed with performance. https://news.samsung.com/global/samsung-unveils-the-galaxy-j2-core-an-introductory-smartphone-packed-with-performance, 2018.

[23] AXBOE, J. Flexible I/O tester. https://github.com/axboe/fio.

[24] BENAVIDES, T., TREON, J., HULBERT, J., AND CHANG, W. The enabling of an Execute-In-Place architecture to reduce the embedded system memory footprint and boot time. *JCP 3*, 1 (2008), 79–89.

[25] BRUMLEY, J. Apple, Samsung continue to lose smartphone market share in shift toward more value. https://seekingalpha.com/article/4101007-apple-samsung-continue-lose-smartphone-market-share-shift-toward-value, 2017.

[26] CHEN, R., WANG, Y., HU, J., LIU, D., SHAO, Z., AND GUAN, Y. Unified non-volatile memory and NAND flash memory architecture in smartphones. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific* (2015), IEEE, pp. 340–345.

[27] HYUN, S., BAHN, H., AND KOH, K. Lecramfs: an efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics 53* (2007).

[28] JEONG, D., LEE, Y., AND KIM, J.-S. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *FAST* (2015), pp. 191–202.

[29] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. AndroStep: Android storage performance analysis tool. In *Software Engineering (Workshops)* (2013), vol. 13, pp. 327–340.

[30] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Framework for analyzing android I/O stack behavior: from generating the workload to analyzing the trace. *Future Internet 5*, 4 (2013), 591–610.

[31] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference* (2013), pp. 309–320.

[32] KANG, D. H., AND EOM, Y. I. FSLRU: a page cache algorithm for mobile devices with hybrid memory architecture. *IEEE Transactions on Consumer Electronics 62*, 2 (2016), 136–143.

[33] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS) 8*, 4 (2012), 14.

[34] KIM, H., RYU, M., AND RAMACHANDRAN, U. What is a good buffer cache replacement scheme for mobile flash storage? In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 235–246.

[35] KIM, J.-M., AND KIM, J.-S. Androbench: benchmarking the storage performance of android-based mobile devices. In *Frontiers in Computer Education*. Springer, 2012, pp. 667–674.

[36] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split.

[37] LEE, K., AND WON, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software* (2012), ACM, pp. 23–32.

[38] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. Usenix.

[39] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi non-volatile RAM for low overhead persistency enforcement in smartphones. In *HotStorage* (2014).

[40] MAHONEY, M. About the test data. http://mattmahoney.net/dc/textdata.html, 2011.

[41] NGUYEN, D. T., ZHOU, G., XING, G., QI, X., HAO, Z., PENG, G., AND YANG, Q. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 287–300.

[42] PARK, H., BAEK, S., CHOI, J., LEE, D., AND NOH, S. H. Exploiting storage class memory to reduce energy consumption in mobile multimedia devices. In *Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference on* (2010), IEEE, pp. 101–102.

[43] REN, J., LIANG, M. C.-J., WU, Y., AND MOSCIBRODA, T. Memory-centric data storage for mobile systems.

[44] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free.

[45] TOLOMEI, S. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2.

[46] ZHANG, X., LI, J., WANG, H., XIONG, D., QU, J., SHIN, H., KIM, J. P., AND ZHANG, T. Realizing transparent OS/Apps compression in mobile devices at zero latency overhead. *IEEE Transactions on Computers 66*, 7 (2017), 1188–1199.

[47] ZHONG, K., WANG, T., ZHU, X., LONG, L., LIU, D., LIU, W., SHAO, Z., AND SHA, E. H.-M. Building high-performance smartphones via non-volatile memory: The swap approach. In *Proceedings of the 14th international conference on embedded software* (2014), ACM, p. 30.

QZFS: QAT Accelerated Compression in File System for Application Agnostic and Cost Efficient Data Storage

Xiaokang Hu
Shanghai Jiao Tong University
Intel Asia-Pacific R&D Ltd.

Fuzong Wang*
Shanghai Jiao Tong University
Intel Asia-Pacific R&D Ltd.

Weigang Li
Intel Asia-Pacific R&D Ltd.

Jian Li
Shanghai Jiao Tong University

Haibing Guan
Shanghai Jiao Tong University

Abstract

Data compression can not only provide space efficiency with lower Total Cost of Ownership (TCO) but also enhance I/O performance because of the reduced read/write operations. However, lossless compression algorithms with high compression ratio (e.g. gzip) inevitably incur high CPU resource consumption. Prior studies mainly leveraged general-purpose hardware accelerators such as GPU and FPGA to offload costly (de)compression operations for application workloads. This paper investigates ASIC-accelerated compression in file system to transparently benefit all applications running on it and provide high-performance and cost-efficient data storage. Based on Intel® QAT ASIC, we propose QZFS that integrates QAT into ZFS file system to achieve efficient gzip (de)compression offloading at the file system layer. A compression service engine is introduced in QZFS to serve as an algorithm selector and implement compressibility-dependent offloading and selective offloading by source data size. More importantly, a QAT offloading module is designed to leverage the vectored I/O model to reconstruct data blocks, making them able to be used by QAT hardware without incurring extra memory copy. The comprehensive evaluation validates that QZFS can achieve up to 5x write throughput improvement for FIO micro-benchmark and more than 6x cost-efficiency enhancement for genomic data post-processing over the software-implemented alternative.

1 Introduction

Data compression has reached proliferation in systems involving storage, high-performance computing (HPC) or big data analysis, such as EMC CLARiiON [14], IBM zEDC [7] and RedHat VDO [18]. A significant benefit of data compression is the reduced storage space requirement for data volumes, along with the less power consumption for cooling per unit of logical storage [12, 51]. Furthermore, if the input data to Hadoop [3], Spark [4] or stream processing job [40] is compressed, the data processing performance can be effectively enhanced as the compression not only saves bandwidth but also decreases the number of read/write operations from/to storage systems.

It is widely recognized that the benefits of data compression come at the expense of computational cost [1, 9], especially for lossless compression algorithms with high compression ratio [41]. In a number of fields (e.g., scientific big data or satellite data), lossless compression is the preferred choice due to the requirement for data precision and information availability [12, 43]. Prior studies mainly leveraged general-purpose hardware accelerators such as GPU and FPGA to alleviate the computational cost incurred by (de)compression operations [15, 38, 41, 45, 52]. For example, Ozsoy *et al.* [38] presented a pipelined parallel LZSS compression algorithm for GUGPU and Fowers *et al.* [15] detailed a scalable fully pipelined FPGA accelerator that performs LZ77 compression. Recently, the emerging AISC (Application Specific Integrated Circuit) compression accelerators, such as Intel® QuickAssist Technology (QAT) [24], Cavium NITROX [34] and AHA378 [2], are attracting attentions because of their advantages on performance and energy-efficiency [32].

Data compression can be integrated into different system layers, including the application layer (most common), the file system layer (e.g., ZFS [48] and BTRFS [42]) and the block layer (e.g., ZBD [27] and RedHat VDO [18]). Professional storage products such as IBM Storwize V7000 [46] and HPE 3PAR StoreServ [19] may contain competitive compression feature as well. If compression is performed at the file system or lower layer, all applications, especially big data processing workloads, running in the system can transparently benefit from the enhanced storage efficiency and reduced storage I/O cost per data unit. This feature also implies that only lossless compression is acceptable to avoid influences on applications. To the best of our knowledge, there is no practical solution at present that provides hardware-accelerated data compression at the layer of local or distributed file systems.

In this paper, we propose QZFS (QAT accelerated ZFS) that

*Co-equal First Author

integrates Intel® QAT accelerator into the ZFS file system to achieve efficient data compression offloading at the file system layer so as to provide application-agnostic and cost-efficient data storage. QAT [24] is a modern ASIC-based acceleration solution for both cryptography and compression. ZFS [6, 48] is an advanced file system that combines the roles of file system and volume manager and provides a number of features, such as data integrity, RAID-Z, copy-on-write, encryption and compression.

In consideration of the goal of cost-efficiency, QZFS selects to offload the costly gzip [1] algorithm to achieve high space efficiency (i.e., high compression ratio) and low CPU resource consumption at the same time. QZFS disassembles the (de)compression procedures of ZFS to add two new modules for integrating QAT acceleration. First, a compression service engine module is introduced to serve as a selector of diverse compression algorithms, including QAT-accelerated gzip and a number of software-implemented compression algorithms. It implements compressibility-dependent offloading (i.e., compression/non-compression switch) and selective offloading by source data size (i.e., hardware/software switch) to optimize system performance. Second, a QAT offloading module is designed to efficiently offload compression and decompression operations to the QAT accelerator. It leverages the vectored I/O model, along with address translation and memory mapping, to reconstruct data blocks prepared by ZFS, making them able to be accessed by QAT hardware through DMA operations. This kind of data reconstruction avoids expensive memory copy to achieve efficient offloading. Besides, considering QAT characteristics, this module further provides buffer overflow avoidance, load balancing and fail recovery.

In the evaluation, we deploy QZFS as the back-end file system of Lustre [44] in clusters with varying nodes, and measure the performance with FIO micro-benchmark and practical genomic data post-processing. For FIO micro-benchmark, QZFS with QAT-accelerated gzip can achieve up to 5x average write throughput with a similar compression ratio (3.6) and about 80% reduction of CPU resource consumption (from more than 95% CPU utilization to less than 20% CPU utilization), compared to the software-implemented alternative. For practical genomic data post-processing workloads, benefiting from QAT acceleration, QZFS provides 65.83% reduction of average execution time and 75.58% reduction of CPU resource consumption over the software gzip implementation. Moreover, as compression acceleration is performed at the file system layer, QZFS also significantly outperforms the traditional simple gzip acceleration for applications while conducting genomic data post-processing.

2 Background and Motivation

This section presents data compression benefits and the motivation of hardware-assisted compression.

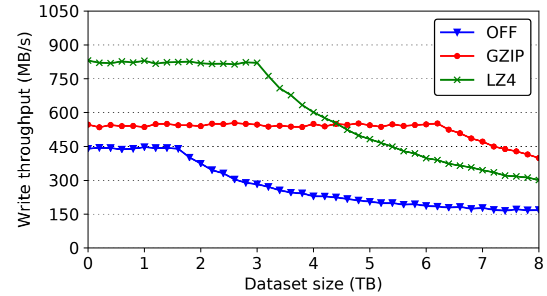


Figure 1: Write throughput on hybrid storage of one 1.6TB NVMe SSD and backup HDDs. Gzip and LZ4 achieve a compression ratio of about 3.8 and 1.9 respectively.

2.1 Data Compression on Storage Devices

As high-performance storage devices, NVMe SSDs can remarkably improve the read/write speed with low energy consumption [25, 49]. Nonetheless, the limited capacity and high price significantly discourage their widespread use and storage devices have accounted for a large proportion of Total Cost of Ownership (TCO) [50]. In the Mistral Climate Simulation System, storage devices occupy more than 20% of the TCO for the entire system [28]. Many studies have investigated data compression on storage devices to improve I/O performance and reduce system TCO simultaneously [31, 36].

To show the benefits of data compression, we evaluated the performance of a compression-enabled file system (i.e., ZFS) by the FIO tool [5] on a hybrid storage system, including one 1.6TB NVMe SSD (Intel® P3700 series) and backup HDDs. Two representative lossless compression algorithms, gzip [16] and LZ4 [35], were used in ZFS to compare with the compression OFF configuration. As shown in Figure 1, the write throughput with data compression (for both gzip and LZ4) outperforms the case of OFF because compression can effectively reduce the total data size written into the storage [33,53]. If the dataset size is larger than the capacity of the 1.6TB NVMe SSD, the excessive data are written into backup HDDs. Due to the poor read/write performance of HDD, the OFF configuration incurs throughput degradation rapidly once the dataset size exceeds 1.6TB. The gzip algorithm achieves a compression ratio of about 3.8 in this evaluation and the write throughput degrades after the dataset size exceeds 6.1TB. Since LZ4 is a fast compression algorithm (i.e. CPU time for compression is largely reduced), it can bring a higher write throughput than the gzip case although the compression ratio is lower, with a value of about 1.9. However, the performance of the LZ4 configuration begin to degrade after the dataset size exceeds 3TB and has no advantage over the gzip algorithm for a dataset size large than 4.5TB. In conclusion, data compression improves space efficiency and allows more data to benefit from the high-performance storage devices.

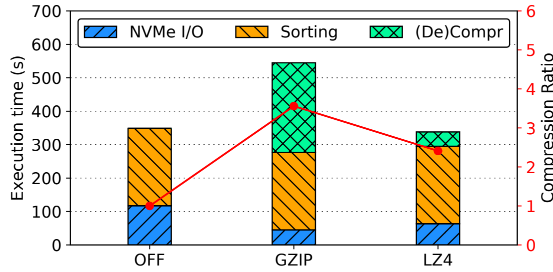


Figure 2: The execution time of genomic data sorting under different compression algorithms

2.2 The Selection of Compression Algorithms

Compression-enabled file systems leverage lossless data compression algorithms to transparently serve upper applications. An algorithm with both high compression ratio and low CPU resource consumption is an optimal choice yet these two aspects are actually hard to achieve at the same time. We conducted an experiment with scientific big data workloads running on ZFS to compare two representative lossless algorithm: gzip and LZ4. SAMtools [30], a popular set of utilities for sequence analysis, was used to manipulate 150GB genomic data stored on a NVMe SSD and perform the sorting operation, which needs to create temporary files and conduct a series of complex data reading and writing actions.

Figure 2 illustrates the breakdown of the total execution time under different compression algorithms, including *NVMe SSD I/O time* (T_{io}), *sorting time* (T_s) and *(de)compression time* (T_c). The execution time for the compression OFF configuration only comprises of T_{io} (117.21s) and T_s (231.77s). The gzip and LZ4 algorithms have a similar T_s value because of the same sorting processing while T_{io} is reduced to 35.02s and 48.45s respectively due to the different compression ratios: 3.56 for gzip and 2.41 for LZ4. However, the high compression ratio of gzip leads to a high T_c value of 278.02s, compared to the 24.77s for the LZ4 configuration. This high CPU resource consumption of gzip may further cause resource competition and impact other tasks. Intuitively, if (de)compression operations can be offloaded to hardware accelerators to eliminate T_c for CPU, gzip could be an ideal compression algorithm as it can achieve the highest space efficiency. This motivates the design of a hardware-assisted compression-enabled file system for high-performance and cost-efficient data storage.

2.3 Hardware-Assisted Data Compression

Nowadays, diverse hardware accelerators are continuously emerging in cloud infrastructures and datacenters [10, 31, 32]. ASIC accelerators are increasingly attracting attentions due to their advantages on performance and energy-efficiency over general-purpose CPU, GPU and FPGA [1, 8, 32]. This paper selects Intel® QAT, a purpose-built ASIC for cryptography and compression, to offload (de)compression tasks and free up CPU resources. The latest high-performance QAT device has been directly integrated into chipsets and is becoming increasingly cheaper [22].

In essence, the offloading of a (de)compression task is to replace the compression-related function call with an I/O call to interact with the underlying QAT accelerator. However, the way the QAT hardware treats data (e.g., physical address and DMA operation) is different from that in the case of software-implemented compression (i.e., using CPU). Runtime translation and optimizations are necessary and important to achieve an efficient offloading. Moreover, considering the offload overhead (e.g., preparation/consumption of QAT requests/responses and PCIe transactions) and the needed preallocated system resources for QAT offloading, not all (de)compression tasks are worth being offloaded into QAT hardware. A well-designed heterogeneous data compression system should investigate the necessity of software/hardware switch or even compression/non-compression switch.

3 System Design

In this paper, based on ZFS file system and Intel® QAT compression accelerator, we propose QZFS (QAT-accelerated ZFS), which can serve as either a local file system or the back-end file system of Lustre (a type of parallel distributed file system). The overall QZFS architecture, including the I/O path of storage and the QAT acceleration subsystem, is illustrated in Figure 3. The modification starts with the ZIO Module of ZFS. The ZIO Module is a centralized I/O process module where all I/O requests are abstracted as ZIOs and forwarded to other modules for further processing, such as data compression and checksum verification. Among the subsequent modules, the ZIO_Compress Module is responsible for data (de)compression. To enable QAT offloading of (de)compression operations, QZFS introduces two new modules: the *Compression Service Engine* and the *QAT Offloading Module*.

To explain the functions of these QZFS modules, the compression workflow is depicted in Figure 3. **(1)-(2)**: The ZIO Module forwards ZIO requests to the ZIO_Compress Module which registers compression algorithms and delivers configuration information to the Compression Service Engine. **(3)**: The Compression Service Engine selects the compression algorithm among one (gzip) accelerated by QAT and others from software compression libraries. **(4)-(5)**: The QAT Offloading Module sends compression requests to the QAT accelerator and consumes QAT responses to fetch compressed result data. **(6)-(8)**: The compressed result data is returned and goes through the upper modules one by one. The data decompression workflow is similar to this compression workflow but does not involve the selection of decompression algorithm.

The main function of the Compression Service Engine is to serve as a selector of diverse compression algorithms. The

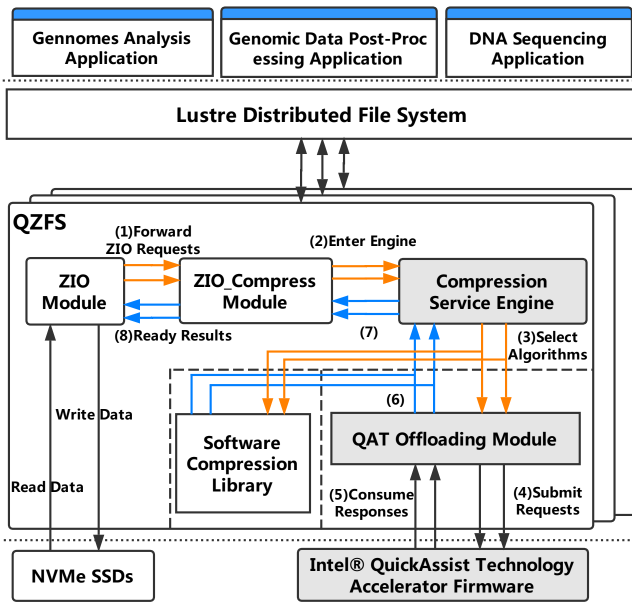


Figure 3: The overall architecture of QZFS

QAT-accelerated gzip is the default algorithm and another four software-implemented compression algorithms, including gzip, LZ4, ZLE and LZJB, are provided. The engine currently selects the compression algorithm according to the configuration set by users. Besides, the Compression Service Engine provides a uniform interface to the upper module. This kind of decoupling makes it able to be easily extended to support other hardware accelerators. Developers only need to focus on the detailed implementation of a new compression scheme and add it to this engine.

The QAT Offloading Module is responsible for offloading data (de)compression operations to the QAT accelerator. The existing source data prepared by the ZIO_Compress Module are suitable for software-based compression schemes. However, they cannot be directly used in the (de)compression requests to the QAT accelerator as data may be mapped into discontiguous physical memory and cannot be accessed through DMA operations of the QAT accelerator. To address this problem, a vectored (a.k.a. scatter/gather) I/O model is introduced to avoid memory copy and ensure that the QAT accelerator can sequentially read source data from multiple flat (i.e., physically contiguous) buffers and organize them to a single data stream for (de)compression, or read (de)compressed result data from a single data stream and organize them to multiple flat buffers.

4 Implementation & Optimization

The implementation of QZFS prototype relies on QAT development APIs and Linux environment (CentOS with Linux Kernel 3.10) and it has been integrated into ZFS official releases [37]. This section introduces detailed features and recent bug fixes of QZFS, especially regarding performance optimizations, to demonstrate the accomplishment of effective compression offloading in QZFS.

4.1 Compression Service Engine

The detailed architecture of the Compression Service Engine is illustrated in Figure 4, which contains a *compressibility checker* and an *algorithm selector*.

4.1.1 Compressibility Dependent Offloading

Compressibility is a performance factor that is determined by data compression ratio to reduce unnecessary offloading operations. Low compressibility means that data are not worth being stored in a compressed format because, in that case, the compressed data will not save much storage space but only incur extra resource consumption for later decompression. Even with the QAT acceleration, the decrease of unnecessary offloading operations is beneficial as QAT resources can be spared for useful (i.e., high compression ratio) tasks, especially in peak hours.

The *compressibility checker* in QZFS uses an auto-rejection method to determine whether to store the data in a compressed format or not. When the checker receives a ZIO request, it has the knowledge of the source data size, denoted by S_{src}, and presets the result data size, denoted by S_{res}, for placing the returned compressed data. Conventionally, S_{res} may be set to the same size as S_{src} when executing compression tasks. In QZFS, the *compressibility checker* uses a default $S_{res} = 0.9 * S_{src}$ for QAT-accelerated gzip algorithm to indicate a compressibility threshold of 10%. If the compressed result data overflows the buffer of S_{res}, it is automatically rejected and the uncompressed source data is returned to the ZIO_Comress Module as the result. Compared to the original compressibility threshold (i.e., 12.5%) in ZFS, QZFS actually relaxes the restriction as the decompression operation can be performed more efficiently by the QAT accelerator. Users can further adjust this compressibility threshold to maximize space efficiency, depending on the characteristics of workloads and hardware conditions. For data decompression, the compressibility checking is not necessary and the S_{res} is set to the size of original uncompressed data, which is recorded by QZFS during compression processing.

4.1.2 Selective Offloading by Source Data Size

The source data size (i.e., S_{src}) is an important factor that may have an influence on system performance. ZFS has a parameter named *record size* which defines the maximum size of a block that may be compressed and written by ZFS. A storage I/O operation with data size smaller than the record size may be packed into one ZIO request for processing. That's to say,

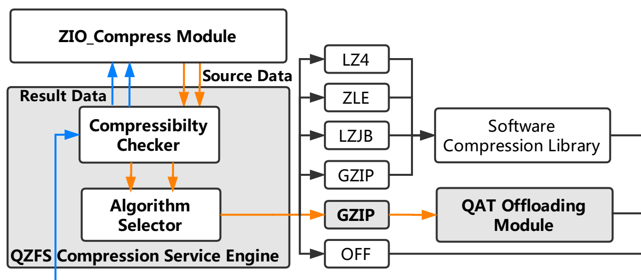


Figure 4: Architecture of compression service engine

S_{src} is variable within the upper limit defined by the record size. ZFS uses a default record size value of 128KB, which achieves a good performance in most cases, and users may modify this parameter to obtain better performance in their own scenarios.

QZFS selectively offloads ZIO requests with S_{src} from 4KB to 1MB and uses software alternatives to process other ZIO requests. For small source data (i.e., $S_{src} < 4\text{KB}$), the offload overhead, including preparation/consumption of QAT requests/responses and PCIe communication, offsets most of the benefit of QAT accelerating, so the software-based compression is a better choice. The QAT support for big source data (i.e., $S_{src} > 1\text{MB}$) requires a large kernel memory preallocated to work as intermediate buffers. This memory size is several times the product of the maximum S_{src} and the number of allocated (typically tens of) QAT instances. Moreover, for a fixed number of worker threads in ZFS that synchronously offload (de)compression operations to QAT, the use of bigger source data cannot give obviously higher performance. Therefore, QZFS only preallocates a kernel memory region that can support a maximum S_{src} of 1MB to achieve both good performance and acceptable consumption of kennel memory resources.

4.1.3 Applicability and Availability

The *algorithm selector* is implemented as a centralized scheduler of data compression algorithms. Algorithms are organized in a scalable algorithm vector that can easily be extended to incorporate other algorithms, either software-based or hardware-assisted ones. Developers only need to focus on the detailed implementation of a new compression scheme, add it to this the algorithm vector and update the configuration as needed. When runtime error occurs in some hardware accelerator, the algorithm selector can seamlessly switch to other software alternatives to provide fault tolerance and high availability.

Currently, the selection of compression algorithm mainly relies the the configuration, which defines the priorities of different algorithms, with the QAT-accelerated gzip as the default one. If in future, a number of hardware-assisted compression algorithms with their own features are incorporated into the Compression Service Engine, an intelligent selection (e.g., use the hints from upper layers) is a good optimization to reap the strengths of different compression schemes.

4.2 QAT Offloading Module

4.2.1 Vectored I/O Model

The QAT accelerator accesses data blocks through DMA operations, which require the data to be stored in contiguous physical memory. The original source and result data of a ZIO request are stored using virtual memory pointed by two pointers P_{src} and P_{res}. The vectored I/O model can effectively bunch together the discontinuous memory to form I/O transactions for DMA operations. As illustrated in the left part of Figure 5, we employ two buffer structures, *flat buffer* and *scatter/gather buffer list* (SGL), to implement the vectored I/O model.

The flat buffer consists of two parts: a data buffer length, denoted by *DataLenInByte* and a pointer, *pData*, to the data buffer owning contiguous physical memory. SGL is introduced to organize multiple flat buffers in a vector manner and consists of four parts: (1)*numBuffers*, the number of flat buffers in this list; (2)*pBuffers*, a pointer to an unbounded array containing multiple flat buffers; (3) *pUserData*, an opaque field; (4)*pPrivateMetaData*, private representation of this buffer list. In summary, SGL describes a collection of flat buffers, each of which is physically contiguous. The QAT accelerator can parse the SGL structure to obtain the beginning physical address of each flat buffer and sequentially access each data block through DMA operations.

4.2.2 Data Reconstruction and Memory Zero Copy

A simple approach for data reconstruction is to allocate enough contiguous physical memory and copy data from/to this memory. Specifically, before the (de)compression offloading, a region of contiguous physical memory is allocated to store data copied from P_{src} and delivered in the request to QAT. Also, another region of contiguous physical memory is allocated to store the response (i.e., result data) from QAT. After the completion of (de)compression offloading, the result data is copied from the contiguous physical memory to P_{res} prepared by ZIO.

Although the allocated contiguous physical memory can be reused by multiple ZIO requests, this approach inevitably introduces the overhead of memory copy, which likely become a bottleneck in today's high speed I/O. Therefore, we introduce the vectored I/O model, along with virtual address translation and memory mapping, to achieve memory zero copy. As shown in Figure 5, the QAT Offloading Module translates the virtual addresses of the source data prepared by ZIO into physical addresses and organizes the physical contiguous data blocks into the Input SGL structure.

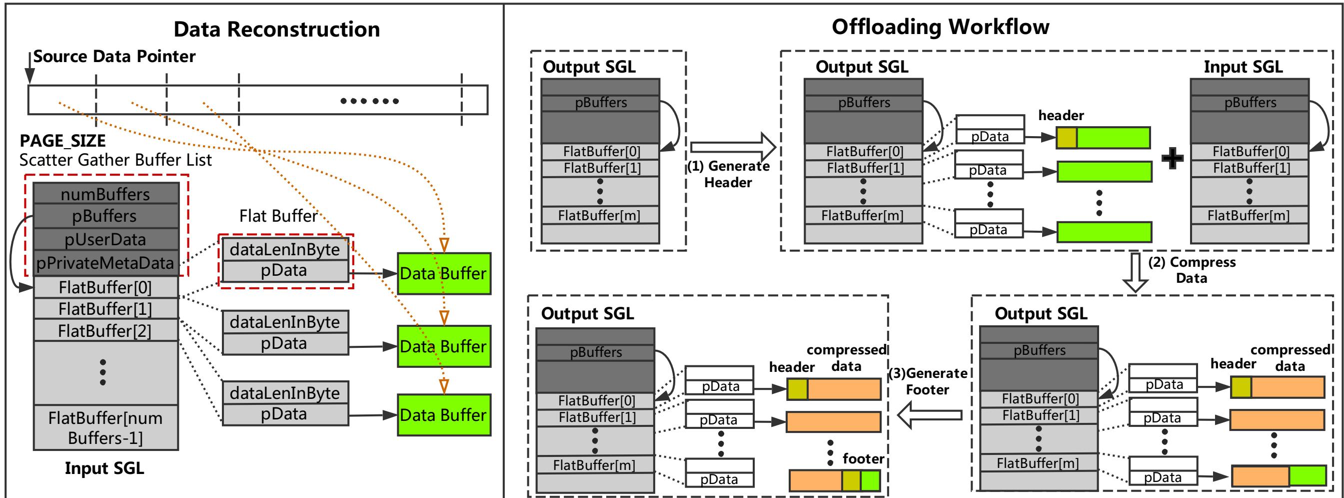


Figure 5: Data reconstruction and QAT offloading workflow in the QAT Offloading Module

A physical page (a.k.a. page frame) is the smallest fixed-length contiguous block of physical memory into which virtual pages are mapped by the operating system. The source data represented by contiguous virtual addresses could be divided into physical contiguous data blocks in terms of physical pages. The QAT Offloading Module directly maps these physical pages to serve as the data of the flat buffers and each flat buffer maps at most one *PAGE_SIZE* data. Note that the start and end virtual addresses of the source data may not be page-aligned, which has an influence on the number of involved physical pages. For example, in the case of 4KB PAGE_SIZE, a 11KB source data may correspond to four physical pages (2KB+4KB+4KB+1KB), instead of three ones. This information is important for the creation of the data structures of Input/Output SGLs. The memory allocation for the Input SGL data structure is performed according to the maximum possible number of flat buffers: $numBuffers = (S_{src} >> \text{PAGE_SHIFT}) + 2$.

The details of the data reconstruction for source data (i.e., building Input SGL) are as follows and the process for the result data buffer to build Output SGL is similar. At first, the virtual address indicated by the P_{src} pointer (pointing to the start address of the source data initially) is translated to obtain the corresponding physical page structure. Note that the virtual address may come from different kernel memory zones, including the *vmalloc region* or the *direct memory region*. The QAT Offloading Module checks whether the a virtual address belongs to the vmalloc region by invoking the *is_vmalloc_addr* function. If so, the *vmalloc_to_page* function is used to get the corresponding physical page structure; otherwise, the *virt_to_page* function is used to obtain the right page. Next, the QAT Offloading Module employs the *kmap* function to establish a long-lasting mapping from kernel's address space to the obtained physical page. The *pData* field of the first flat buffer points to the returned mapped address, plus the same page offset as the P_{src} value. The *dataLenInBytes* field is accordingly set by considering the page offset. Finally, the P_{src} pointer moves to the beginning of the remaining untreated source data and the above steps are repeated to fill subsequent flat buffers. For the last piece of the source data that may correspond to only part of a physical page, the *dataLenInBytes* field is set to the actual size of this last piece. After the completion of an offloading task, the *kunmap* function needs to be invoked to release long-lasting mappings.

4.2.3 QAT Offloading Organization

Overflow avoidance and load balancing: When QZFS boots up, the QAT Offloading Module initializes the QAT logical instances to set up communication channels for requests/responses to/from the QAT accelerator. Thus, a region of contiguous physical memory needs to be allocated for the QAT instance which includes an *intermediate buffer* to place run-time process data (e.g., dynamic Huffman encoding) of the QAT accelerator. The size of the intermediate buffer should be enough for the maximally allowed source data (i.e., 1MB). Data compression is supposed to reduce data size but compression algorithms may cause data expansion at some moment during compression. To avoid the buffer overflow, its size is enlarged to be double of the maximally allowed data size. Besides, when too many data compression tasks are offloaded, the module may not obtain the QAT logical instances because there are not enough QAT computing resources. Therefore, the module will balance the system's computational resources by sending the task to software alternatives and employing the CPU to finish it.

Fail recovery: A QAT compression session describes the compression parameters to be applied across a number of requests. After the initialization of logical instances and source/result data reconstruction, the module sets up the com-

munication with the QAT accelerator by QAT compression sessions. Occasionally, the error may occur in these compression sessions, such as a driver process crash where a wrong DMA address passed to the accelerator. If a failure on the QAT accelerator cannot be handled correctly, the QAT device may be restarted for recovery. In this situation, the QAT Offloading Module cleans all sessions and sets an availability flag as FALSE to disable all QAT offloading actions until the completion of re-initialization. The related internal data structures, QAT logical instances and intermediate buffers will be reset as well.

Offloading workflow: The data compression offloading operations are organized as three steps shown in the right part of Fig. 5. **First**, a header generator API function is called to produce the gzip style header[1], which requires the output SGL as the parameter. The header is added to the front of the output SGL, and the *pData* of the flat buffer is moved to the corresponding offset of the gzip header. **Second**, the data compression API function submits input and output SGLs to the QAT accelerator that consumes the source data from the input SGL and generates processed result data to the output SGL. Note that the QAT accelerator uses the interrupt mode for response processing which requires the module to use the *wait_for_completion_interruptible_timeout* function to wait the completion of the tasks. **Third**, a gzip compliant footer[2] is produced by the footer generator API function. More details of the header and footer formats are given in RFC 1952 [13], which are supported by the QAT accelerator. The difference with data decompression is that it invokes the data decompression API function for three steps.

5 Evaluation

In this section, we first describe the evaluation platform and testing methodology. Then we use FIO micro-benchmark and scientific big data processing workloads to evaluate our implemented QZFS prototype on cluster servers.

5.1 Evaluation Methodology

Experimental testbed: We established an experimental testbed with four physical servers, each of which was equipped with two 22-core Intel® Xeon® E5-2699 v4 processors, 128GB RAM, one NVMe SSD array and one Intel® DH8950 PCIe QAT card [21]. The NVMe SSD array was comprised of three 1.6TB Intel® P3700 Series NVNe SSDs. In each server, a separate SATA SSD was used for housing the operating system (CentOS 7.2) with Linux Kernel 3.10 and QZFS. All these servers were connected via Intel® XL710 40GbE NICs and a 100GbE switch. The detailed topology is illustrated in Figure 6.

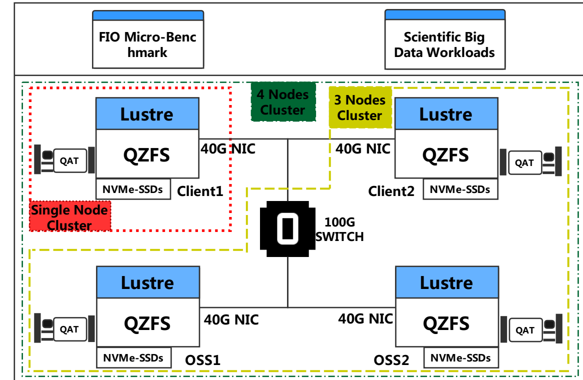


Figure 6: Topology of experimental testbed

Cluster: QZFS was deployed as the back-end file system of Lustre distributed file system. To evaluate different Lustre scenarios, three types of cluster settings were used: (1) all-in-one *single-node cluster*, i.e., a single physical server as both the client and the Object Storage Server (OSS); (2) *three-node cluster* with one physical server as the client and the other two physical servers as OSSes; (3) *four-node cluster* with one more physical server as a client (on the basis of the three-node cluster). All the servers in a cluster shared the same Lustre distributed file system. The client server ran benchmark workloads locally and read/wrote data from/to one or two OSSes.

Note that Lustre inherently provides the ability to support a large number of clients. In our in-lab evaluation, the number of available clients was limited. As a workaround, we leveraged high-performance physical servers along with high-speed NICs to work as *heavy* clients and produce enough stress. In the following experiments, we will show that the total stress is equivalent to tens to hundreds of ordinary clients. The I/O stress from an ordinary client depends on a lot of factors, including workload characteristics, client hardware limits (e.g., NIC limit), the interconnection between Luster clients and OSSes, etc. We assume that each ordinary client generates an I/O stress between 100Mbps and 1Gbps.

Performance metrics: We mainly compared three configurations: (1) *OFF*: QZFS without data compression; (2) *GZIP*: QZFS with software-implemented gzip algorithm enabled; (3) *QAT*: QZFS with QAT-accelerated gzip algorithm enabled. Other algorithms in the software compression library including LZ4, ZLE and LZJB may also be measured for comparison. The following important metrics were collected as performance indicators: read/write *throughput* of micro-benchmarks, average *execution time* of completing big data analytic tasks, *compression ratio* for indicating space efficiency and *CPU utilization* (collected using Intel® Performance Analysis Tool [23]) for indicating computing resource

[1] a gzip header indicates metadata including compression ID, file flags, timestamp, compression flags and operating system ID.

[2] a gzip footer containing a CRC-32 checksum and the length information of the original uncompressed data.

consumption.

Particularly, we define a new comprehensive metric to measure the *cost-efficiency* of the entire system, which equals compression ratio divided by CPU utilization (i.e., computing resource consumption):

$$\frac{compression_ratio}{cpu_utilization}$$

CPU and high-performance storage devices account for a large proportion of TCO. This metric reflects a combined benefit of saved storage space and saved CPU resources, and a higher value means more cost-efficiency gains. Note that we do not consider QAT cost in this metric because QAT device is becoming increasingly cheaper. The latest high-performance QAT, directly integrated into chipsets, only costs about $32 (comparing $132 for C625 chipset with QAT and $100 for C624 chipset without QAT) [22], which is negligible in comparison to CPU price.

5.2 Evaluation Benchmark

Micro-benchmark workloads: FIO (Flexible I/O tester) [5] is a productive tool in Linux that can simulate all kinds of I/O workloads and accurately measure I/O performance. We used FIO to spawn several threads or processes for measuring read/write throughput. The default FIO setting for random data generation and management were employed. More specifically, a buffer of random data (actually pseudo-random) was created at the beginning and it was used continuously during the test to reduce overhead. To comprehensively evaluate QZFS, we conducted the FIO experiments with different I/O patterns, block sizes (i.e, the size of chunks for issuing read/write I/O) and compression algorithms.

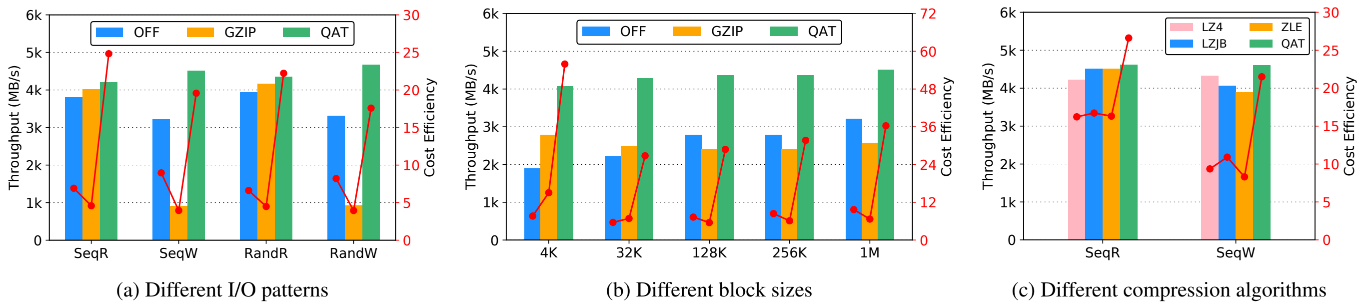
Scientific big data workloads: Genomic data post-processing is a representative workload of scientific big data. The original 3TB genomic dataset used in this experiment are available in European Nucleotide Archive [29] and International Genome Sample Resource [11]. These data are stored as specific formats, such as FastQ, BAM and SAM, which are widely used in both industry and academia. For each client, two genomic data post-processing tools, SAMTools (v1.3.1) with htslib (1.3.2) [30] and Biobambam2 (v2.0.82) with libmaus (2 2.0.435) [47], were used to work as computing workloads. Specifically, we selected five representative operations from these two tools to evaluate the advantage of QZFS. *Converting* is to convert one kind of genomic data to other kind (e.g. FastQ format converted to BAM format in this evaluation), which involves heavy CPU-bound tasks. *Viewing* is to print all alignments information in the specified input file to standard output in SAM format (with no header). *Sorting* is to sort these data by leftmost coordinates and create temporary BAM files as needed when the entire alignment data cannot fit into memory. *Merging* is to merge multiple sorted files, producing a single sorted output file that contains all the input records and maintains the existing sorting order. *Indexing* is to index a coordinate-sorted BAM or CRAM file for fast random access.

5.3 FIO Micro-benchmark

The experiments were conducted in the four-node cluster where 16 FIO threads were created in each Lustre client to read/write job files independently from/to the two Lustre OSSes. The total size of file I/O for each FIO thread is 2GB. The read/write throughput stated in the evaluation is the sum value collected from two clients and the CPU utilization is the average value collected from Lustre OSSes.

Figure 7a shows experimental results for diverse I/O patterns with a fixed 128KB FIO block size, including sequential read/write (SeqR/SeqW) and random read/write (RandR/RandW). In the compression OFF configuration, the average read throughput (i.e., the average value of SeqR and RandR) is 18% higher than the average write throughput, and RandR achieves up to 3937 MB/s throughput, which is the highest. These results are in accordance with the hardware characteristics of NVMe SSDs [17]. The average read throughput in the GZIP configuration outperforms the average write throughput by about 4.5x because the gzip algorithm has an asymmetric compression/decompression speed, with the former lagging much behind the latter [1]. After enabling the QAT accelerator, the average read throughput is similar with the average write throughput. In general, the QAT configuration has the highest read/write throughput and cost-efficiency. The GZIP configuration has a similar compression ratio with the QAT configuration (3.78:3.65), but its high CPU resource consumption (i.e., long compression time) causes not only low write throughput but also low cost-efficiency. The QAT offloading for gzip compression operations achieves about 5x improvement on average write throughput and enhances the cost-efficiency by a factor of more than four. In comparison to the OFF configuration, the QAT configuration also provides a throughput improvement (10% for read and 28% for write) as the compressed data reduces storage I/O cost. Meanwhile, the cost-efficiency is enhanced by a factor of more than three due to the high compression ratio (3.65).

Figure 7b shows the I/O throughput and cost-efficiency with the same amount of sequential read and write operations (SeqR:SeqW = 1:1) while varying the FIO block size in each I/O request from 4KB to 1MB. Note that the record size (128KB by default) in ZFS only defines the maximum size of a block that may be compressed and written by ZFS. As a result, a FIO block may be directly compressed/written by ZFS or multiple FIO blocks may be combined into a whole block for processing. For the compression OFF configuration which can directly benefit from the high-performance NVMe SSDs, the I/O throughput gradually grows from 1899MB/s to 3213MB/s (about 70% improvement) as the increase of FIO block size. For the GZIP and QAT configurations, the



(a) Different I/O patterns (b) Different block sizes (c) Different compression algorithms

Figure 7: The read/write throughput and cost-efficiency on FIO micro-benchmark

varying of FIO block size does not show obvious influence on I/O throughput. Still, the QAT configuration always has the highest I/O throughput and cost-efficiency. A notable phenomenon is that for the case of 4KB FIO block size with software or QAT-accelerated gzip in ZFS, the cost efficiency (actually compression ratio) is obviously higher than other cases with bigger FIO block sizes. This may be a joint result of the record size mechanism in ZFS and the reuse of random data in FIO. Specifically, multiple small (i.e., 4KB) FIO blocks may have a higher probability of being combined into a whole block in ZFS for compression. Then, the possible reuse of random data across these FIO blocks leads to a high compression ratio for the whole block.

Figure 7c compares the QAT-accelerated gzip with other software-implemented fast compression algorithms including LZE, LZJB and ZLE. The QAT configuration gains an average of 12.71% throughput improvement with SeqW and an average of 4.83% throughput improvement with SeqR, compared to these software-implemented algorithms. Also, QAT-accelerated gzip in ZFS provides an average of 2.25x and 1.62x cost-efficiency for SeqW and SeqR respectively. This performance advantage comes from the high compression ratio of the gzip algorithm and the offloading of gzip (de)compression operations. In addition, we can see that SeqR operations show an obviously higher cost-efficiency than SeqW operations. This is because decompression typically costs less computing resources in comparison to compression.

Finally, we give a calculation about how many ordinary clients the total stress in FIO experiments is equivalent to. As shown in Figure 7, the average stress from the two heavy client servers is more than 4000MB/s in the QAT configuration, which equals the stress from 32 to 320 ordinary clients (100Mbps to 1Gbps each).

5.4 Scientific Big Data Evaluation

The scientific workloads running on Luster clients interact with the Luster OSSes to access the stored scientific data. We used two different deployment modes for evaluation: shared

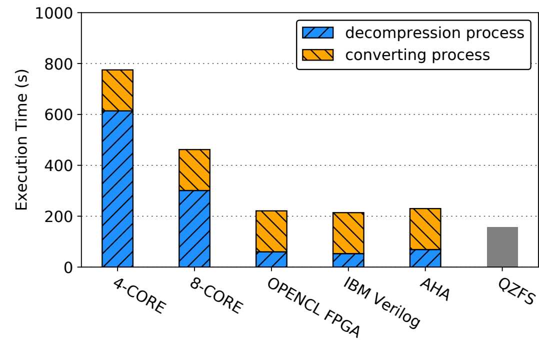


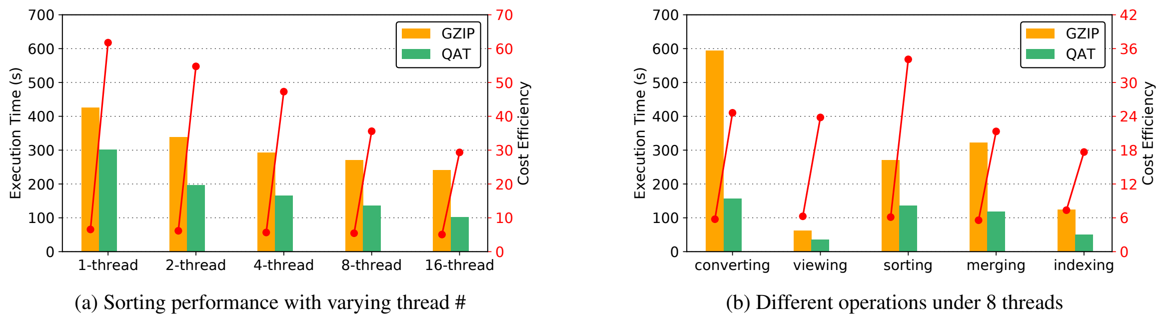
Figure 8: Execution time of a complete converting operation under different data compression schemes

deployment mode (i.e., single-node cluster) and separate deployment mode (including three-node and four-node clusters).

5.4.1 Shared Deployment Mode

Eight scientific workload processes ran in parallel in the shared server. Each workload process was set to read/write 9.5GB data and a total of 76GB data may be processed simultaneously in memory.

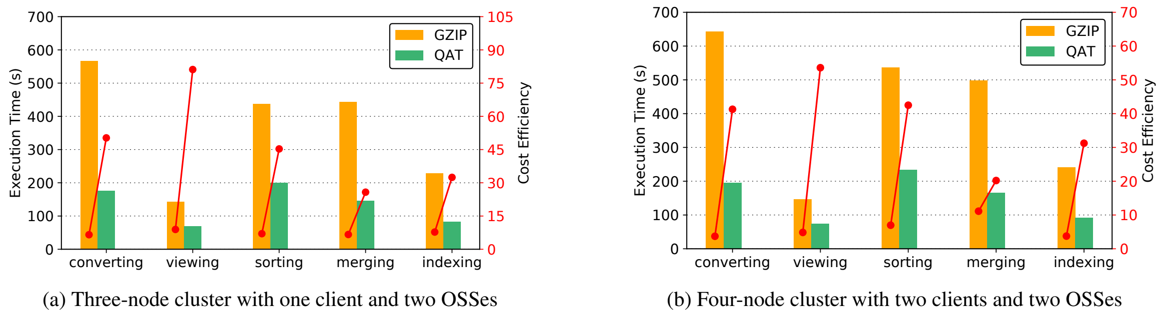
We first validated the benefits of QAT-accelerated gzip compression at the file system layer over *simple* gzip use at the application layer (i.e., the compressed big data needs to be first decompressed into storage and then read again for processing). The experiment results are shown in Figure 8. For the first five configurations, the execution time of a complete converting operation consists of two parts: (1) decompression process time with given computing resources (e.g., 4 cores or AHA accelerator), which includes the time for reading the compressed data from storage, the time for decompression and the time for writing the uncompressed data into storage; (2) converting process time using eight scientific workload processes, which includes the time for reading uncompressed data from storage (76GB totally), the time for converting operations and the time for writing new format data back to storage.



(a) Sorting performance with varying thread #

(b) Different operations under 8 threads

Figure 9: The execution time and cost-efficiency on shared mode deployment (single-node cluster)



(a) Three-node cluster with one client and two OSSes

(b) Four-node cluster with two clients and two OSSes

Figure 10: The execution time and cost-efficiency on separate mode deployment

The first five configurations share a same converting process time of 161.12s. The decompression process time of OpenCL FPGA, IBM Verilog and AHA is calculated according to the performance results reported in [1]. These accelerators show an obvious advantage on decompression compared to the 4 or 8 CPU cores. QZFS keeps data compression transparent to scientific workloads and obtains the shortest execution time (156.71s). This value is even smaller than the above pure converting process time without (de)compression (161.12s) because QZFS directly reads compressed data from storage, performs decompression and converting simultaneously and then writes compressed data back to storage. Since the reading/writing of compressed data largely reduces storage I/O cost and (de)compression is offloaded to QAT accelerator, a higher performance (i.e., shorter execution time) is achieved.

It is true that an application may integrate (de)compression module that can efficiently process compressed big data (e.g., small block based (de)compression and multiple threads), along with the enabling of QAT acceleration, to achieve similar performance to QZFS. However, this likely involves heavy modifications for each new application. In comparison, one-time modification to the file system (i.e., the proposed QZFS) can transparently benefit all applications running on it.

Figure 9a shows the execution time and cost-efficiency with varying number of threads in each scientific workload process performing sorting operations. When there is only one thread in each workload process, the software gzip configuration gives an execution time of 426s and a cost-efficiency value of 6.6. The QAT configuration provides a 30% reduction of execution time and enhances the cost-efficiency by a factor of nearly 10 due to the faster (de)compression and largely reduced CPU resource consumption. As the thread number increases from 1 to 16, the execution time decreases gradually for both GZIP and QAT configurations. In the case of 16 threads in each workload process, the performance advantage of QAT-accelerated gzip grows from 30% to 60% as more threads produce more parallel I/O requests and increases the utilization of underlying QAT accelerators. This further leads to a growing of CPU utilization for sorting from 5.8% to 12.2% in the QAT configuration. In comparison, the CPU utilization in the GZIP configuration grows from 64% to 83%.

Figure 9b evaluates the post-processing scientific workloads performing different operations with a fixed eight threads in each workload process. All the five types of operations need to read the genomic data from storage and the operations excluding viewing further needs to write the newly-generated data (smaller or bigger) back to storage. We can see that the QAT configuration achieves 73% and 63% reduction of execution time for the converting and merging operations respectively over the GZIP configuration. For other opera-

tions, the QAT-accelerated gzip in ZFS can provide about 2x performance enhancement. On average, the QAT configuration brings a 3.91x cost-efficiency improvement over the software-implemented gzip. Especially, the cost-efficiency enhancement is up to 5.57x for the sorting operation because it is a complicated operation that creates many intermediate data files and repeatedly involves reading (decompression) and writing (compression) actions.

The total stress from the one heavy client (eight threads in each workload process) is up to 75GB/35s=2143MB/s (the viewing operation), which equals the stress from 17 to 171 ordinary clients (100Mbps to 1Gbps each).

5.4.2 Separate Deployment Mode

We first evaluated QZFS with one client that remotely accesses two OSSes and then increased the client number to two. In each client sever, eight scientific workload processes ran in parallel and each workload process with a fixed eight threads was set to read/write 9.5GB data.

In the three-node cluster with different operations, as shown in Figure 10a, the QAT configuration on average reduces 63.10% execution time and achieves more than 6x cost-efficiency compared with the QAT configuration. Like the shared deployment mode, the QAT-accelerated gzip in ZFS still shows a high performance enhancement (68.95% reduction of the execution time) for the converting operation. A notable phenomenon is that the biggest cost-efficiency enhancement (9.11x) is witnessed on the viewing operation because it does not need to write data back to the remote storage (i.e., reduced CPU resource consumption on the costly network stack).

For the case of four-node cluster with double stress from two heavy clients, as shown in Figure 10b, the overall performance results are similar to the case of three-node cluster. In comparison to the software-implemented gzip in ZFS, the QAT acceleration on average provides a 63.14% reduction of execution time and a 6.26x improvement of cost-efficiency. It demonstrates that QZFS has a good scalability to provide stable and effective compression services as the scaling of clients. The total stress from the two heavy clients is up to 150GB/75s=2000MB/s (the viewing operation), which equals the stress from 16 to 160 ordinary clients (100Mbps to 1Gbps each).

6 Bottleneck Analysis

This section analyzes the performance bottleneck in QZFS. We use the experiment result of RandW in Figure 7a for analysis, which shows the highest I/O throughput (4680MB/s) for the QAT configuration. In this case, the average CPU utilization in the two OSSes is 20.2%, which means CPU resources are still abundant. The compression ratio is 3.55 and the actual NVMe SSD I/O throughput is 4680/3.55 = 1318MB/s, which means the NVMe SSD is not the bottleneck as the compression OFF configuration can achieve a storage I/O throughput of 3314MB/s. The network throughput for each physical server is about 4680/2 = 2340MB/s = 18.72Gpbs, which is only half of the hardware limit of 40GbE NIC. The QAT compression throughput in each Lustre OSS is also about 18.72Gpbs, which reaches nearly 80% of the hardware limit (24Gpbs [21]) of a DH8950 QAT card. In summary, the I/O throughput (4680MB/s) does not achieve the hardware limit of the testbed system while this throughput cannot be further enhanced even if we launch more FIO threads in clients to generate more RandW jobs in parallel.

Actually, the main performance bottleneck may reside in the ZFS software stack. Although ZFS is designed to automatically leverage multi-core resources, the number of ZFS worker threads that can offload (de)compression operations to QAT is restricted by: the number of CPU cores and the number of available QAT instances. What's more, the worker thread interacts with QAT in a synchronous mode, which means the worker cannot submit the next (de)compression request until the completion and consumption of the first one. As a result, the use of more FIO threads at the application layer cannot give rise to more concurrent/parallel (de)compression requests sent to QAT.

An approach to overcome this bottleneck is to optimize the way the worker thread interacts with QAT to increase the utilization of the underlying QAT accelerator. QAT provides an inherently non-blocking interface with the request/response mechanism. If we can enable asynchronous offload mode in ZFS, a single worker thread then has the ability to concurrently offload multiple (de)compression operations to QAT and the QAT hardware limit can be easily reached with only several threads. However, the enabling of asynchronous offload mode is a hard work, involving the design of (1) asynchronous support in all layers of ZFS software stack to correctly handle an uncompleted (de)compression block and (2) efficient pause (context saving) and resumption (context restoring) of an offload job. One can refer to our previous work [20] on how to enable high-performance asynchronous crypto offload framework for TLS servers/terminators.

7 Related Work

Hardware-assisted data compression techniques has been well studied in the literature by using general-purpose accelerators. Patel *et al.* proposed a parallel algorithm and implemented a bzip2-like lossless data compression scheme on GPU architecture [39]. Their implementation enabled the GPU to become a co-processor of data compression, which lightened the computing burden of CPU by using idle GPU cycles. Ozsoy *et al.* [38] presented a pipelined parallel LZSS compression algorithm for GUGPU. Li *et al.* proposed an efficient, scalable GPU-accelerated OLAP system which tackled the bandwidth discrepancy using compression and an optimized

data transfer path [31]. Abdelfattah *et al.* utilized the Open Computing Language to implement high-speed gzip compression on an FPGA, which achieved higher throughput over standard compression benchmark, with equivalent compression ratio [1]. Kim *et al.* presented high throughput Xpress FPGA compressor to achieve CPU resource saving and high power efficiency [26]. Fowers *et al.* detailed a scalable fully pipelined FPGA accelerator that performs LZ77 compression and static Huffman encoding at rates up to 5.6 GB/s [15]. Pekhimenko *et al.* employed FPGA and GPU accelerators to implement Base-Delta encoding data compression algorithms in stream processing [40]. In comparison, our QZFS leverages the emerging ASIC accelerator for compression offloading and integrates it into the layer of file system to provide transparent, high-performance and cost-efficient compression service.

8 Conclusions

High storage I/O performance and low Total Cost of Ownership (TCO) are two important optimization objectives, which are hard to be obtained at the same time. Data compression is considered as an effective solution, but the compression tasks incur high computing resource consumption and likely impact the running of application workloads. Instead of directly accelerating compression tasks in applications, this paper investigated the data compression offloading at the file system level. QZFS (QAT accelerated ZFS) was proposed to integrate Intel® QAT accelerator into ZFS file system to provide application-agnostic and cost-efficient data storage. The evaluation has validated that QZFS can effectively save CPU resources and further enhance the performance of big data processing workloads in comparison with the software-implemented gzip in ZFS or traditional gzip acceleration for applications.

Acknowledgments

The authors would like to thank the shepherd, Dr. Patrick P. C. Lee, and the anonymous reviewers for their valuable comments. This work is supported in part by the National Key Research and Development Program of China (No. 2016YFB1000502) and the National Science Fund for Distinguished Young Scholars (No. 61525204). Jian Li is the corresponding author.

References

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL (IWOCL)*, pages 4:1–4:9, 2014.

[2] AHA. Aha378: 80.0 gbps gzip compression/decompression accelerator, 2016. http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=41.

[3] Apache. Apache hadoop, 2018. https://hadoop.apache.org/.

[4] Apache. Apache spark: Lightning-fast unified analytics engine, 2018. https://spark.apache.org/.

[5] Jens Axboe. Welcome to fio's documentation, 2017. https://fio.readthedocs.io/en/latest/.

[6] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST)*, volume 215, 2003.

[7] Paolo Bruni, Maria Kroos Boisen, Gianmauro De Marchi, and Franco Pinto. Reduce storage occupancy and increase operations efficiency with ibm zenterprise data compression. Technical report, 2018.

[8] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 7:1–7:13, 2016.

[9] Yanpei Chen, Archana Ganapathi, and Randy H Katz. To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pages 23–28, 2010.

[10] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO)*, pages 225–236, 2010.

[11] Laura Clarke, Susan Fairley, Xiangqun Zheng-Bradley, Ian Streeter, Emily Perry, Ernesto Lowy, Anne-Marie Tassé, and Paul Flicek. The international genome sample resource (igsr): A worldwide collection of genome variation incorporating the 1000 genomes project data. *Nucleic acids research*, 45(D1):D854–D859, 2016.

[12] Constantinos Costa, Georgios Chatzimilioudis, Demetrios Zeinalipour-Yazti, and Mohamed F Mokbel. Efficient exploration of telco big data with compression and decaying. In *Proceedings of the 33rd International Conference on Data Engineering (ICDE)*, pages 1332–1343, 2017.

[13] Peter Deutsch. Gzip file format specification version 4.3. Technical report, 1996.

[14] EMC. Emc data compression: A detailed review. Technical report, 2010. https://www.emc.com/collateral/hardware/white-papers/h8045-data-compression-wp.pdf.

[15] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 23rd International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 52–59, 2015.

[16] Jean-Loup Gailly and Mark Adler. The gzip home page, 2018. https://www.gnu.org/software/gzip/.

[17] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.

[18] Christian Horn. A look at vdo, the new linux compression layer, 2018. https://www.redhat.com/en/blog/look-vdo-new-linux-compression-layer.

[19] HPE. Hpe 3par storeserv storage, 2019. https://www.hpe.com/us/en/storage/3par.html.

[20] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. Qtls: high-performance tls asynchronous offload framework with intel® quickassist technology. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–172, 2019.

[21] Intel. Product brief: Intel® quickassist adapter 8950. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf, 2015.

[22] Intel. Compare intel® products. https://ark.intel.com/content/www/us/en/ark/compare.html?productIds=97341,97342, 2019.

[23] Intel. Intel® platform analysis technology. https://software.intel.com/en-us/intel-platform-analysis-technology, 2019.

[24] Intel. Intel® quickassist technology (intel® qat), 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[25] 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 *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[26] Joo Young Kim, Scott Hauck, and Doug Burger. A scalable multi-engine xpress9 compressor with asynchronous data transfer. In *Proceedings of the 22rd International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 161–164, 2014.

[27] Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. Transparent online storage compression at the block-level. *ACM Transactions on Storage (TOS)*, 8(2):5:1–5:33, 2012.

[28] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Data compression for climate data. *Supercomputing Frontiers and Innovations*, 3(1):75–94, 2016.

[29] Rasko Leinonen, Ruth Akhtar, Ewan Birney, Lawrence Bower, Ana Cerdeno-Tárraga, Ying Cheng, Iain Cleland, Nadeem Faruque, Neil Goodgame, Richard Gibson, et al. The european nucleotide archive. *Nucleic acids research*, 39(suppl_1):D28–D31, 2010.

[30] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[31] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[32] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: specializing the datacenter. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 178–190, 2016.

[33] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 1–14, 2010.

[34] Marvell. Nitrox iii security processor family, 2019. https://www.marvell.com/security-solutions/nitrox-security-processors/nitrox-iii/index.jsp.

[35] Takayuki Matsuoka. Lz4 - extremely fast compression, 2019. https://lz4.github.io/lz4/.

[36] Sangwhan Moon, Jaehwan Lee, Xiling Sun, and Yang-suk Kee. Optimizing the hadoop mapreduce framework with high-performance storage devices. *The Journal of Supercomputing*, 71(9):3525–3548, 2015.

[37] OpenZFS. Zfs hardware acceleration with qat, 2018. http://open-zfs.org/wiki/ZFS_Hardware_Acceleration_with_QAT.

[38] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel lzss for streaming data compression on gpgpus. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 37–44, 2012.

[39] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. Parallel lossless data compression on the gpu. In *Proceedings of the Innovative Parallel Computing (InPar)*, 2012.

[40] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. Tersecades: efficient data compression in stream processing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 307–320, 2018.

[41] Weikang Qiao, Jieqiong Du, Zhenman Fang, Libo Wang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *Proceedings of the 26th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, 2018.

[42] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, 2013.

[43] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[44] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, pages 380–386, 2003.

[45] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A Ross. Massively-parallel lossless data decompression. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP)*, pages 242–247, 2016.

[46] Jon Tate, Christian Burns, Bosmat Tuv-El, and Jorge Quintal. Ibm real-time compression in ibm san volume controller and ibm storwize v7000. Technical report, 2018.

[47] German Tischler and Steven Leonard. biobambam: tools for read pair collation based algorithms on bam files. *Source Code for Biology and Medicine*, 9(1):13, 2014.

[48] Wikipedia. Zfs, 2019. https://en.wikipedia.org/wiki/ZFS.

[49] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Manu Awasthi, Tameesh Suri, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 473–474, 2015.

[50] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. A fresh perspective on total cost of ownership models for flash storage in datacenters. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 245–252, 2016.

[51] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 111–124, 2016.

[52] Bin Zhou, Hai Jin, and Ran Zheng. A high speed lossless compression algorithm based on cpu and gpu hybrid platform. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 693–698, 2014.

[53] Hongbo Zou, Yongen Yu, Wei Tang, and Hsuan-wei Michelle Chen. Improving i/o performance with adaptive data compression for big data applications. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1228–1237, 2014.

Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies

Youngseok Yang[1] Jeongyoon Eo[1] Geon-Woo Kim[2] Joo Yeon Kim[3]
Sanha Lee[4] Jangho Seo[1] Won Wook Song[1] Byung-Gon Chun[1]*
[1]*Seoul National University* [2]*Viva Republica* [3]*Samsung Electronics* [4]*Naver Corp.*

Abstract

Optimizing scheduling and communication of distributed data processing for resource and data characteristics is crucial for achieving high performance. Existing approaches to such optimizations largely fall into two categories. First, distributed runtimes provide low-level policy interfaces to apply the optimizations, but do not ensure the maintenance of correct application semantics and thus often require significant effort to use. Second, policy interfaces that extend a high-level application programming model ensure correctness, but do not provide sufficient fine control.

We describe Apache Nemo, an optimization framework for distributed dataflow processing that provides fine control for high performance, and also ensures correctness for ease of use. We combine several techniques to achieve this, including an intermediate representation, optimization passes, and runtime extensions. Our evaluation results show that Nemo enables composable and reusable optimizations that bring performance improvements on par with existing specialized runtimes tailored for a specific deployment scenario.

1 Introduction

It is becoming increasingly important to optimize scheduling and communication for different characteristics of resources and data in distributed data processing. Examples of such characteristics widely discussed in recent literature are geographically-distributed resources [19, 33, 44, 45], cheap transient resources [37, 38, 42, 47, 48], disk-based large data shuffle [35, 36, 51], and skewed data [22, 24, 25, 34]. Researchers have shown that the existing scheduling and communication methods, unaware of these characteristics, often suffer from substantial performance degradation.

Distributed runtimes such as Dryad [20], Tez [40], and the Spark runtime [4] provide low-level interfaces to plug in computation scheduler and data channel policies to optimize for such diverse deployment scenarios. These policy interfaces have direct access to control messages and data elements, and can apply optimizations such as placing computations on specific types of resources and performing in-memory data shuffle. Unfortunately, runtime policy developers must exercise care to ensure that the policies they build and apply maintain correct application semantics. The main reason is that runtime interfaces are designed to be general, and allow

*Corresponding author.

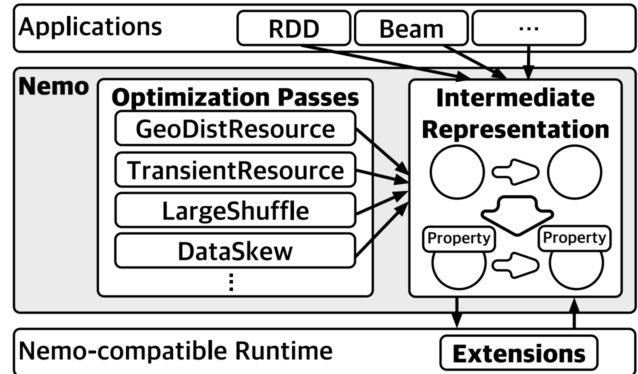


Figure 1: Nemo optimizes scheduling and communication of distributed data processing.

for arbitrary modifications to scheduling and communication methods.

On the other hand, policy interfaces integrated with a high-level application programming model offer indirect control over runtime execution. For example, Optimus [22] integrates with the DryadLINQ programming model to enable specifying alternative DryadLINQ subqueries. This ensures correct application semantics as long as the specified subqueries compute the same results, and thus reduces the effort required to build different optimization policies. However, such application-level interfaces do not provide sufficient fine control over distributed scheduling and communication, because application programming models are designed to hide distributed execution from application developers.

To overcome the limitations of existing interfaces, we believe it is critical to introduce a new policy interface that provides both fine control for high performance, and also ensures correct application semantics for ease of use. In this work we take a middle ground between the existing runtime and application-level interfaces. We design a policy interface that transforms an intermediate representation (IR) of applications to express indirect but fine-grained control over distributed scheduling and communication.

There are three main challenges to designing an optimization framework that embodies this middle ground approach. First, the framework should define the IR transformation methods that provide fine control and also ensure correctness. Second, the framework should enable the development of reusable and composable user-defined optimization policies

that transform the IR. Third, the framework should apply the transformations of the IR in the distributed execution of the application.

Figure 1 depicts our Nemo optimization framework that addresses the challenges. Specifically, its IR directed-acyclic graph (DAG), optimization passes, and runtime extensions address the three challenges, respectively. Nemo integrates with high-level application programming model libraries, and compatible distributed runtimes.

First, the Nemo IR DAG represents a data processing application with vertices representing logical operations and edges representing data dependencies. To ensure that the transformed IR DAG produces the same outputs as the original IR DAG, we provide two types of transformation methods: reshaping and annotation. Reshaping methods can insert a set of utility vertices whose semantics are known to Nemo, such as a vertex that samples data. Annotation methods set execution properties of each vertex and edge to configure fine-grained scheduling and communication, such as speculative cloning and data persistence strategies. Nemo ensures correctness using the information about the communication patterns (e.g., shuffle) of edges, and the information about the configured utility vertices and execution properties.

Second, the Nemo optimization pass abstraction enables expressing optimizations as a function that takes as input an IR DAG and calls its transformation methods. Because a pass is a simple function, different combinations of passes can be applied across different applications. We show that optimization techniques previously employed in specialized runtimes, such as Iridium [33] and Pado [48], can be expressed as optimization passes with concise lines of code.

Third, the Nemo runtime extensions integrate with the underlying runtime to apply the IR DAG transformations. Runtimes typically provide a runtime DAG abstraction to run computations on a cluster of machines [4, 20, 40]. Our scheduler extension applies various scheduling policies when scheduling the IR vertices of an IR DAG through a runtime DAG. It also rewrites the runtime DAG during job execution to apply run-time optimizations. Our data channel extension applies the optimized data communication within the runtime DAG.

We have implemented Nemo, and also a distributed runtime that is compatible with Nemo. At present, Nemo provides full support for Beam [1] applications and a subset of Spark RDD [50] applications. Our runtime integrates with REEF [46] to run on Hadoop YARN [2] and Mesos [18] clusters. We have evaluated Nemo in a cluster of Amazon EC2 instances using different optimization passes, datasets, and resource environments. Evaluation results show that each optimization pass brings performance improvements on par with existing specialized runtimes, and combinations of passes further improve performance for scenarios with a combination of different resource and data characteristics. Nemo is currently an Apache Incubator project [3].

```
class TreeAggregate implements ConnectionManager {
 void onUpstreamVertexEvent(event) {
  mapVertexGroups = analyzeLocationsAndSizes(event)
  aggregateVertices = newVertices(mapVertexGroups)
  connect(mapVertexGroups, aggregateVertices)
 }
}

class Repartition implements ConnectionManager {
 void onUpstreamVertexEvent(event) {
  desiredPartitions = analyzeDataStatistics(event)
  modifyPartitionVertices(desiredPartitions)
  modifyReduceVertices(desiredPartitions)
 }
}
```

Figure 2: Pseudocode of Dryad policies. The Dryad policy interface provides fine control over distributed scheduling and communication, but does not ensure correctness.

2 Background

We first discuss in detail the existing runtime policy interfaces and application-level policy interfaces using concrete code examples. Specifically we describe the interfaces of Dryad [20] and Optimus [22].

The Dryad policy interface allows for arbitrary modifications to its directed-acyclic graph (DAG) representation of applications. In a Dryad DAG, a vertex represents a unit of work performed on a machine and an edge represents a data transfer from a vertex to another. For example, a map-reduce application can be represented in Dryad as a number of map vertices fully connected with a number of reduce vertices. The Dryad runtime coordinates the scheduling and communication of the vertices on a cluster of machines.

Figure 2 shows the pseudocode of two example Dryad policies [5]. Here, `ConnectionManager` is a callback-based abstraction that listens to events from the configured upstream vertices. First, `TreeAggregate` builds an aggregation tree with a goal to use network bandwidth resources more efficiently. Suppose `TreeAggregate` listens to the map vertices in a map-reduce application, to obtain the information on the locations and sizes of map vertex outputs. Using the information, `TreeAggregate` groups map vertices, creates intermediate aggregation vertices, and then connects each map vertex group to an aggregation vertex. Second, `Repartition` dynamically distributes data with a goal to handle data skew. Suppose the map-reduce application additionally has bucketizer vertices that consume sample output data from the map vertices, and partition vertices that partition the original map vertex outputs prior to transferring the data to the reduce vertices. Then, `Repartition` can be used to monitor the bucketizer vertices, and modify the partition and reduce vertices with the goal to evenly distribute the map outputs.

```
// Application code
mulA = defineMatMulSubqueryA(matrixX, matrixY)
mulB = defineMatMulSubqueryB(matrixX, matrixY)

// Policy code
stats = collectDataStatistics(matrixX, matrixY)
rewriter.registerAlternatives(stats, mulA, mulB)
```

Figure 3: Pseudocode of an Optimus policy. The application-level Optimus policy interface ensures correctness, but provides coarse-grained control of substituting subqueries.

As shown by these examples, runtime policies can configure various scheduling and communication methods.

However, the flexibility of runtime interfaces comes at a cost: the policy developer must exercise care to ensure application correctness when developing, reusing, and composing different policies [4, 20, 22, 40]. First, the interface allows for a bug in `TreeAggregate` to miss connecting one of the map vertices to an intermediate aggregation vertex, making the optimized DAG produce partial results. Second, `Repartition` can break application semantics when applied on a random vertex in a different DAG that does not use bucketizer and partition vertices. Third, applying both `TreeAggregate` and `Repartition` on the same DAG can lead to conflicting executions that produce incorrect results. Manually building a combined policy can require a significant effort for complex policies, such as the `DrDynamicAggregateManager` in Dryad that consists of 1.3K lines of C++ code [5]. As a consequence, runtime policies have been mostly hard coded in runtimes and data processing application compilers such as the DryadLINQ compiler [22,49], and the Hive compiler [43]. The authors of Optimus also report that their system-level optimization policies are hard-coded in the DryadLINQ compiler, maintaining the DAG property and operator semantics for the pre-defined operators in DryadLINQ [22].

In contrast to runtime interfaces, Optimus provides an application-level policy interface that ensures correctness, by restricting the interface to substituting DryadLINQ subqueries. Figure 3 shows the pseudocode for optimizing a matrix multiplication application described in the original Optimus paper [22]. The code defines two alternative subqueries for multiplying two matrices, and a policy for selecting a subquery to use for the execution. Note that as long as the two subqueries produce the same results, changing the policy code does not alter the semantics of the application. However, as this example shows, such application-level policy interfaces lack fine-grained control over scheduling and communication like selecting the types of resources to run specific computations on. The main reason is that application programming models are designed to hide distributed execution from application developers.

3 System Design

The goal of the Nemo optimization framework is to support fine control over distributed execution of data processing applications, and at the same time maintain correct application semantics. Concretely, given a *DAG* representation of a data processing application with deterministic operations and a user-defined policy P where $DAG' = P(DAG)$, Nemo aims to provide the following properties.

- **Correctness**: Given the same inputs the optimized DAG' should produce the same outputs as the *DAG*, even when P is applied while the *DAG* is being executed. This ensures that the optimizations maintain correct application semantics.
- **Reusability**: The same P should be applicable to different *DAG*s. This enables reusing the same policy across different data processing applications, although the effects may differ between applications.
- **Composability**: If P and P' do not override optimizations specified by the other policy then enable composing different policies like $P'' = (P \circ P')$. If the policies do have a conflict, then automatically detect it for analysis. This enables distinct policies that each optimizes for a different resource or data characteristic to be incorporated into a single policy.

We show how Nemo combines an intermediate representation (IR) DAG, optimization passes, and runtime extensions to ensure these properties. First, the IR DAG provides reshaping and annotation methods for specifying optimizations (Section 3.1). Second, optimization passes define functions that operate on the IR DAG methods (Section 3.2). Third, runtime extensions apply the optimizations in the underlying runtime (Section 3.3).

3.1 Intermediate Representation

The Nemo IR DAG aims to provide the desired *DAG* representation of an application. The main challenge in designing the IR DAG is defining the methods for transforming it. For Nemo to ensure the desired properties, we make explicit both the intention and the effect of the optimization for each method invocation. For example, instead of providing a single method to insert arbitrary computations, we provide multiple higher-level methods such as those specifically for increasing parallelism, speculative cloning, and sampling. We describe the IR DAG reshaping and annotation methods that embody this approach, and in particular how those methods enable ensuring correctness. We then discuss the types of applications and runtimes supported by our IR DAG design.

IR DAG Reshaping: irdag.insert()	$Relay(f : x \to x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \to v), oneToOne(v \to e.dst)\}$
	$Reshuffle(f : x \to x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \to v), shuffle(v \to e.dst)\}$
	$Sampling(f : x \to sv.f(x)), sv, rate$	:	$V \cup \{v\}, E \cup \{e.comm(e.src \to v) \mid e \in E \wedge e.dst = sv\}$
	$Trigger(f : x \to udf(x)), udf, e$	:	$V \cup \{v\}, E \cup \{oneToOne(e.src \to v)\}$
	(V/E = original vertex/edge set, v = inserted vertex, f = function of v, $e.comm$ = oneToOne/shuffle/broadcast)		
IR Vertex Annotation: v.set()	$Parallelism/Integer$	:	sets the number of tasks for executing v
	$SpeculativeCloning/Thresholds$	:	sets the thresholds for determining and cloning straggler tasks
	$ResourceSite/Map(Index, Site)$	:	sets the geographical sites of the resources to place tasks on
	$ResourcePriority/Enum(Transient)$	:	sets the priority of the resources to place tasks on
IR Edge Annotation: e.set()	$DataFlow/Enum(Pull, Push)$	:	$e.dst$ is scheduled after $e.src$ finishes, or scheduled concurrently
	$DataStore/Enum(Memory, Disk)$	:	$e.src$ tasks store output data for e in memory, or disk
	$NumPartitions/Integer$	:	sets the number of partitions that $e.src$ tasks create for e
	$PartitionSets/List(Set(Index))$	:	sets the partitions that each $e.dst$ task fetches for e
	$Persistence/Enum(Keep, Discard)$	:	sets whether to keep or discard data after $e.dst$ processes e

Table 1: Example IR DAG transformation methods for optimizing scheduling and communication. Reshaping methods take as input a utility vertex and additional arguments. Annotation methods take as input a key/value execution property.

3.1.1 Transforming an IR DAG

The Nemo IR DAG represents a data processing application with vertices representing logical operations and edges representing data dependencies. When executed, an IR vertex is translated into parallel tasks that run on multiple nodes. An IR edge can be translated into key-partitioned data blocks that are produced by tasks. The initial IR DAG translated from an application, such as an RDD [50] and Beam [1] application, typically consists of vertices containing functions defined by the application, and edges with the information on communication patterns (one-to-one, shuffle, broadcast).

Table 1 shows example reshaping and annotation methods Nemo provides to transform the IR DAG. The reshaping methods specify a new utility vertex to insert into the IR DAG, and Nemo inserts new edges to connect the specified vertex with the existing vertices in the IR DAG. Table 1 specifies four utility vertices. `Relay` and `Reshuffle` simply apply an identity function to forward data from an upstream vertex to a downstream vertex, connecting with the downstream vertex with the one-to-one and the shuffle dependency, respectively. `Sampling` vertex applies the same function as an existing vertex, and consumes the same data that the existing vertex consumes. During the execution, Nemo schedules only a subset of `Sampling` tasks according to the given sampling rate. `Trigger` vertex applies a user-defined function on intermediate data. When a `Trigger` vertex executes and completes, Nemo collects the results of the user-defined function to generate a message. Nemo then halts the execution of the job, and uses the message to trigger a corresponding run-time optimization pass, which we describe in Section 3.2. The IR DAG also supports deleting the inserted utility vertices.

The annotation methods configure scheduling and communication of vertices and edges by annotating specified execution properties. Table 1 specifies nine execution properties. For scheduling, we have execution properties for deciding how, where, and when to schedule tasks. `Parallelism` and `SpeculativeCloning` configure how many tasks to schedule. `ResourceSite` and `ResourcePriority` specify where to schedule the tasks. `DataFlow` determines whether or not to schedule source and destination tasks concurrently. For communication, we enable configuring the medium to store intermediate data with `DataStore`, the persistence method with `Persistence`, and data partitioning strategies with `NumPartitions` and `PartitionSets`. Combinations of different execution properties can express optimizations that can require significant efforts to implement with runtime policy interfaces. For example, we can configure upfront task cloning with a persistent in-memory data shuffle that pushes data eagerly from transient resources to reserved resources, through simply annotating appropriate `SpeculativeCloning`, `ResourcePriority`, `Persistence`, `DataStore`, and `DataFlow` properties on two vertices and a shuffle edge that connects them. The IR DAG also supports looking up the execution properties annotated on vertices and edges.

3.1.2 Ensuring Correctness

The reshaping methods ensure correctness, because Nemo connects the newly inserted utility vertex with existing vertices correctly. As shown in Table 1, only the outputs of the `Relay` and `Reshuffle` vertices are consumed by existing vertices, and these outputs are equivalent to the data that the existing vertices originally consumed. The other utility vertices, on the other hand, do not reach data sinks and thus do not affect the final results that the IR DAG produces. When a utility vertex is specified to be deleted, Nemo reverts appropriate changes.

The annotation methods ensure correctness through enabling Nemo to examine the configured execution properties. For each vertex in the IR DAG, Nemo checks its execution

properties and the execution properties of its neighboring edges and vertices, while also examining the communication patterns of the edges. This ensure correctness because execution properties do not use and modify computation semantics [17, 21, 52] inside each vertex, and also do not have direct access to control messages and data elements in the runtime. For example, Nemo checks that the sets in the `PartitionSets` are disjoint and together contain all offsets for the `NumPartitions`, to read each partition exactly once. Nemo also checks that `PartitionSets` and `NumPartitions` are set on shuffle edges, and that vertices connected with an one-to-one edge have the same `Parallelism`. `Persistence`, for example, is not checked, because discarded intermediate data can always be recomputed from the source data when needed.

Our transformation methods ensure correctness even when invoked during the execution of the IR DAG. Because the IR DAG is decoupled from the underlying runtime, Nemo ensures correctness by controlling when to apply the transformations of the IR DAG in the runtime. Specifically, we define that a vertex is being executed when its tasks are being executed, and an edge is being executed when its source or destination vertex is being executed. First, if the transformed vertices and edges have not yet been executed, then we apply the changes immediately, such that the changes are used when they are executed. Second, if they are being executed, then we delay applying the changes until they finish execution to ensure correctness. Third, if they have already finished execution, then we apply the changes immediately, such that the changes are used when they are re-executed due to reasons such as faults.

3.1.3 Supported Applications and Runtimes

The current design of the IR DAG supports data processing applications that can be represented as a DAG of data-parallel and deterministic operators that process bounded data. Many real-world applications, such as Beam and RDD batch applications and also higher-level domain-specific applications like machine learning and SQL applications, meet this assumption. The current IR DAG would need to be extended to support other types of applications, such as those that have cyclic dependencies and process unbounded data [31].

The IR DAG assumes an underlying distributed runtime that supports configuring and applying utility vertices and execution properties. Existing runtimes can be enhanced to provide full support for the IR DAG optimizations through introducing additional features. For example, new data channels in addition to the existing ones (FIFO, File, TCP Pipe) can be introduced in Dryad [20] to provide support for various combinations of the `DataStore`, `DataFlow`, and `Persistence` execution properties. Similarly, a feature to dynamically add computations to a running application can be introduced in Tez [40] and the Spark runtime [4] to apply utility vertices inserted at run time.

3.2 Optimization Passes

Nemo optimization passes aim to provide the desired user-defined policy abstraction P. A pass is a function that receives an input IR DAG and produces a transformed IR DAG. We first describe how to develop and compose passes. We then describe how Nemo applies the given passes on the IR DAG.

3.2.1 Developing and Composing Passes

We describe the rationale and the algorithm for several example passes to demonstrate how to develop and compose new passes. We can write two types of passes: compile-time and run-time. Compile-time passes take as input only an IR DAG, and are run prior to job execution. Run-time passes additionally receive a message produced by a `Trigger` vertex during job execution.

Geo-distributed data analytics: We aim to cope with the low and variable capacity of WAN links when processing data that are geographically distributed [19, 33, 44, 45]. To reduce network bottlenecks, we formulate the problem of placing computations to geographically distributed sites as a linear program (LP), similar to specialized scheduler extensions like Iridium [33]. Here, we use bandwidth information and data size estimations. We also use an off-the-shelf linear solver library, since Nemo allows using external libraries when writing a pass. The pseudocode of this algorithm is as follows.

```
CompileTimePass GeoDistPass(irdag):
 solution = solveLP(bwInfo(), sizeEstimates(irdag))
 for v in irdag.vertices:
  v.set(newResourceSite(solution.get(v)))
```

Harnessing transient resources: We aim to reduce recomputation costs when using transient resources that are cheap but frequently evicted [37, 38, 42, 47, 48]. Based on the communication patterns, we identify operations that incur large recomputation costs and place them on reserved resources. We place the other operations on transient resources. We also quickly move intermediate data produced on transient to reserved resources. This applies key scheduling and communication optimizations employed in specialized runtimes like Pado [48]. The pseudocode of this algorithm is as follows.

```
CompileTimePass TransientResourcePass(irdag):
 for v in irdag.vertices.topologicallySorted():
  if (allOneToOneFromReserved(v.inEdges)
   || existsNonOneToOne(v.inEdges)):
   v.set(ResourcePriority.Reserved)
  else:
   v.set(ResourcePriority.Transient)
  for e in v.inEdges:
   if fromTransientToReserved(e.src, v):
    e.set(DataFlow.Push)
```

Large-scale data shuffle: We aim to reduce random disk read overheads that can grow quadratically with data size when shuffling data, similar to specialized shuffle systems

like Sailfish [35] and Riffle [51]. We insert a Relay vertex to specify shuffling data in memory as soon as produced and writing the data as-is to a local disk. We also ensure that the in-memory data are discarded once transferred, to avoid running into out of memory errors. Following computations sequentially read the data from the local disk, after the shuffle completes. The pseudocode of this algorithm is as follows.

```
CompileTimePass LargeShufflePass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()):
  rv = newRelayVertex()
  irdag.insert(rv, e)
  rv.inEdge.set(DataFlow.Push, DataStore.Memory)
  rv.inEdge.set(Persistence.Discard)
  rv.outEdge.set(DataFlow.Pull, DataStore.Disk)
```

Mitigating data skew: We aim to assign the same amount of data across parallel computations to prevent stragglers. We first set the number of partitions for the data to be shuffled. We then insert a Trigger vertex with a function for obtaining the set of data partition sizes. We also ensure that the shuffle receiver is executed after the the shuffle sender and the Trigger vertex complete, at which point we will have obtained the statistics and optimized the execution of the shuffle receiver. The pseudocode of this algorithm is as follows.

```
CompileTimePass SkewCTPass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()):
  e.set(newNumPartitions(e), DataFlow.Pull)
  irdag.insert(newOptVertex(), sizeFunction(), e)
```

At run time, when the Trigger vertex completes and makes available the set of size numbers, we partition the set into subsets such that the sum of the numbers in the subsets are as equal as possible. We then assign each subset to a distinct shuffle receiver task. The pseudocode of this algorithm is as follows.

```
RunTimePass SkewRTPass(irdag, message):
 subsets = partition(message)
 message.edge.set(newPartitionSets(subsets))
```

Finally, we can compose multiple passes to build an optimization policy like the following example. Registering a run-time pass requires specifying a compile-time pass that inserts Trigger vertices, which produce the same type of message the run-time pass uses.

```
policyBuilder.register(LargeShufflePass)
policyBuilder.register(SkewRTPass, SkewCTPass)
policy = policyBuilder.build()
```

3.2.2 Applying Passes

Given an IR DAG and a policy composed of passes, Nemo first applies the compile-time passes on the IR DAG in the same order as they were registered. The optimized IR DAG output by the last compile-time pass is executed. As the execution progresses, each Trigger vertex completes execution and produces a message. For each message, Nemo runs the corresponding run-time pass to transform the IR DAG. Nemo runs the passes for different messages serially.

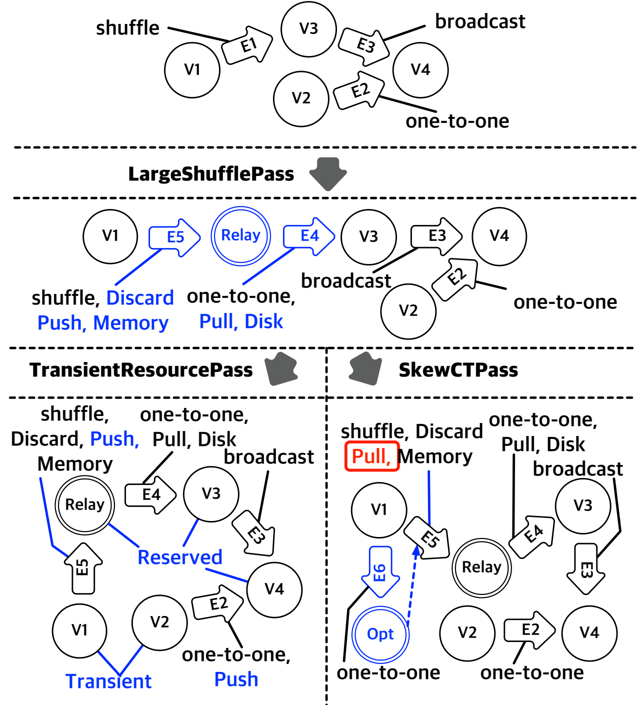


Figure 4: A policy composed of the LargeShufflePass and the TransientResourcePass, and another policy composed of the LargeShufflePass and the SkewCompileTimePass are applied on an input IR DAG.

After applying each pass, Nemo checks whether the IR DAG produced by the pass is correct as described in Section 3.1.2, and also whether the pass has encountered a conflict with a previous pass. A conflict occurs when a pass overwrites the value of an execution property set by a previous pass to a different value, or deletes a utility vertex inserted by a previous pass. Nemo throws an error and refuses to execute in case of a check failure after running a compile-time pass. Upon a check failure of a run-time pass, Nemo just ignores the IR DAG output by the pass and logs the failure, as stopping an already running application can be costly.

Figure 4 shows how Nemo runs two example policies. Both policies first apply the LargeShufflePass, which inserts a Relay vertex between V1 and V3, and annotates E5 and E4. The first policy then applies the TransientResourcePass, which performs annotations without any conflict with the previous pass. The second policy applies the SkewCTPass, which inserts a Trigger vertex, and tries to annotate E5 with the pull DataFlow. However, the SkewCTPass encounters a conflict as the push DataFlow has already been set for E5 by the previous LargeShufflePass.

Fundamentally, the conflict in the second policy occurs

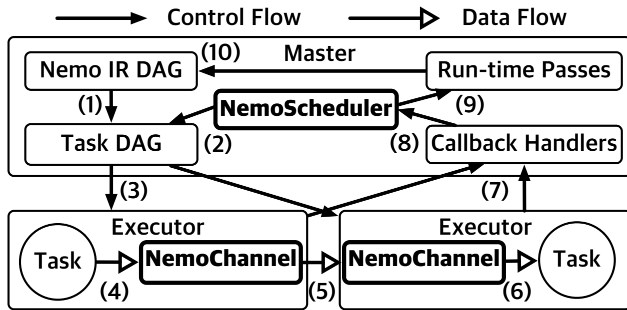


Figure 5: Nemo runtime extensions (bold) apply optimizations in a distributed runtime.

because the `LargeShufflePass` tries to shuffle data eagerly in memory, whereas the `SkewCTPass` tries to use the statistics of the data before the downstream computations start to consume the data. If undetected, this conflict results in a pull-based in-memory data shuffle, where the outputs of all `V1` tasks are stored in memory before the `Relay` tasks start fetching the data. Although this configuration avoids disk seek overheads and also handles data skew at the same time, it can cause out of memory errors for large input data.

Because Nemo detects such conflicts explicitly, we can quickly address the issue. In this case, we design a new `SkewSamplingPass` that avoids the conflict with the `LargeShufflePass`. This new compile-time pass clones the IR DAG using `Sampling` vertices, and first runs the clone to obtain the statistics of sampled data. Our third policy with the `LargeShufflePass` and the `SkewSamplingPass` can be applied together on the IR DAG to optimize for both large data shuffle and data skew. However, compared to the `SkewCTPass`, the `SkewSamplingPass` incurs the cost of executing additional vertices and using the statistics of sampled data rather than the entire data.

Next, we describe how these various transformations of the IR DAG are reflected in the distributed execution.

3.3 Runtime Extensions

We use a Nemo-compatible runtime depicted in Figure 5 to describe how the Nemo runtime extensions apply the IR DAG transformations in the distributed runtime. Upon job launch, the runtime starts a master process and executor processes on user-specified resources. In the master, the `NemoScheduler` extension operates on the task DAG abstraction that the runtime provides for scheduling tasks to executors. Executors spawn a thread to run each scheduled task, and uses the `NemoChannel` extension to communicate data between the tasks. In the rest of the section we describe how these extensions apply optimizations.

First, we set up the initial task DAG using the IR DAG optimized by compile-time passes (1). Here, we merge neighboring IR vertices into the same tasks as much as possible to minimize data communication overheads, while considering communication patterns of the IR edges and related execution properties such as the `Resource` properties and the `Parallelism` property. In case of a `Trigger` vertex, we also register a callback handler to collect the results produced by the corresponding tasks from executors as a message. Upon job start, we select candidate tasks for scheduling, which are the source tasks and their children tasks connected with the push `DataFlow` (2). For each candidate task, we select candidate executors by comparing the corresponding `Resource` properties of the task with the information on the executors. We then schedule the task to a candidate executor with the least number of running tasks (3).

When a task emits a data element, we write it to the corresponding `DataStore` implementation, creating a data block when all data elements for the channel are written (4). If the corresponding edge is shuffle, then the block is partitioned into `NumPartitions`. When a task reads input data elements, we look for the locations of the input data blocks, blocking the call when looking for blocks that are not yet available. We fetch the input data elements from the local and remote `DataStores`, while applying `PartitionSets` for shuffle edges (5-6). Once all of the downstream tasks successfully read a block, we decide to either keep or discard the block based on the `Persistence` property.

Upon learning about task progress and executor status, we schedule new tasks, restart tasks to recover from failures and evictions, and clone tasks based on the `SpeculativeCloning` property (7-8). When a message is produced for a `Trigger` vertex, we postpone scheduling new tasks, invoke the corresponding run-time pass (9), rewrite the task DAG based on the new IR DAG output by the run-time pass at the correct timing described in Section 3.1.2 (10), and resume scheduling.

4 Implementation

We have implemented Nemo and a distributed runtime that is compatible with Nemo in around 32K lines of Java code. Our Nemo implementation consists of the following three components similar to Musketeer [15] and LLVM [26]: frontend, optimizer, and backend.

The frontend translates applications such as Beam and RDD applications into an IR DAG (Section 3.1). At present, our frontend provides translation support for all Beam [1] operators, and a subset of `RDD` [50] operators such as `map`, `reduce`, `collect`, `broadcast`, and `cache`. The main reason for not fully supporting RDDs is that the current iterator implementation used in Nemo is not readily compatible with the various `RDD` implementations. In the future we plan to modify our iterator implementation to address this limitation. The optimizer applies optimization passes on the IR DAG (Section 3.2). The backend configures the underlying runtime

with the optimizer and the runtime extensions (Section 3.3).

Existing Beam applications can run on Nemo by modifying the line importing the Beam `PipelineRunner` implementation to our implementation of the runner. The frontend converts each Beam `PTransform` to an IR vertex, and `PCollection` to an IR edge. The frontend also obtains the information on communication patterns during the translation. For example, it specifies shuffle edges for the incoming `PCollections` of the `GroupByKey` `PTransforms`.

Similar to Beam, existing RDD applications can run on Nemo with simple modifications to the lines importing the implementations of `SparkSession` and `SparkContext` to our implementations of the classes. Each RDD becomes an IR edge, and each user-defined function that generates an RDD becomes an IR vertex. Our frontend also aims to respect all of the user-specified parameters on RDDs such as parallelism and data caching, by setting the execution properties on the translated IR DAG accordingly.

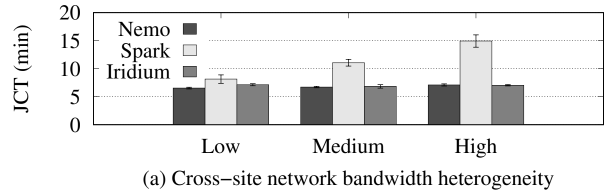
Our runtime implementation is built on top of REEF [46], and consists of master and executor processes similar to the Nemo-compatible runtime described in Section 3.3. A REEF job consists of the driver that obtains containers from a resource manager, and evaluators that provide runtime environments on containers. To take advantage of the abstractions provided by REEF, the runtime master runs as the REEF driver and the runtime executors run as the REEF evaluators. Through the integration with REEF [46], our runtime runs on resource managers such as Hadoop YARN [2] and Mesos [18].

5 Experimental Evaluation

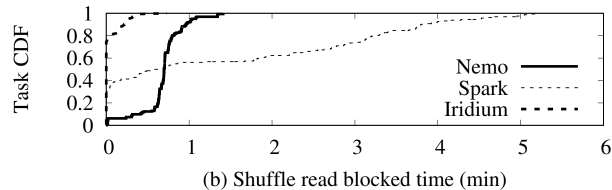
We evaluate Nemo on the following three dimensions. First, we evaluate how Nemo applies fine control under different resource and data characteristics. Second, we evaluate how different combinations of optimization passes optimize the same application. Third, we evaluate how the same Nemo policy optimizes different applications.

We run data processing applications with different combinations of following resource and data characteristics: geographically distributed resources, transient resources, large-shuffle data, and skewed data. We run each application five times, and we report the mean values with error bars showing standard deviations.

We use h1.4xlarge Amazon EC2 instances, each of which provides 16 vCPUs, 64 GiB memory, two 2 TB HDDs, and 10 Gbps network. We use different numbers of instances for different experiments. On each instance, one of the two disks is used by a Hadoop Distributed File System [2] cluster that we set up on the instances, and the other is used as a scratch disk for maintaining intermediate data. Input datasets are stored in HDFS, and fetched by the systems at the beginning of each job.



(a) Cross−site network bandwidth heterogeneity



(b) Shuffle read blocked time (min)

Figure 6: JCT for different cross-site network bandwidths, and CDF of shuffle read blocked time of tasks under the high cross-site network bandwidth heterogeneity.

5.1 Fine Control

In this experiment, we evaluate how Nemo applies fine control under different resource and data characteristics. For comparison we run Spark 2.3.0 [4], because it is an open-source, state-of-the-art system. We also run a specialized runtime for each deployment scenario. Specifically, we run Iridium [33] for geo-distributed resources, Pado [48] for transient resources, and Hurricane [12] for data skew. We examine the results of Beam applications on Nemo and Pado, Spark RDD applications on Spark and Iridium, and a Hurricane application on Hurricane.

We confirm that the baseline performance is comparable for Beam and basic RDD applications on Nemo. We also confirm that the baseline performance is comparable for Spark and Nemo with the `DefaultPass`, which configures pull-based on-disk data shuffle with locality-aware computation placement similar to Spark. We observe that the overhead of running the compile-time passes on Nemo is roughly 200ms. We also measure and report run-time overheads of the `Relay` vertex, `Trigger` vertex, and `SkewRTPass` in this section.

Geo-Distributed Resources: To set up geo-distributed resources and heterogeneous cross-site network bandwidths, we use Linux Traffic Control [6] to control the network speed between instances, as described in Iridium [33]. Each site is configured with 2Gbps uplink network speed, and a specific downlink network speed between 25Mbps and 2Gbps. We experiment with Low, Medium, and High bandwidth heterogeneity with the fastest downlink outperforming the slowest downlink by 10×, 41×, and 82×. With this, we use 20 EC2 instances as resources scattered across 20 sites. To evaluate data shuffle under heterogeneous network bandwidths, we use a workload that joins two partitions of 373GB Caida [8] network trace dataset and computes network packet flow statistics.

The job completion time (JCT) of Iridium, Spark, and

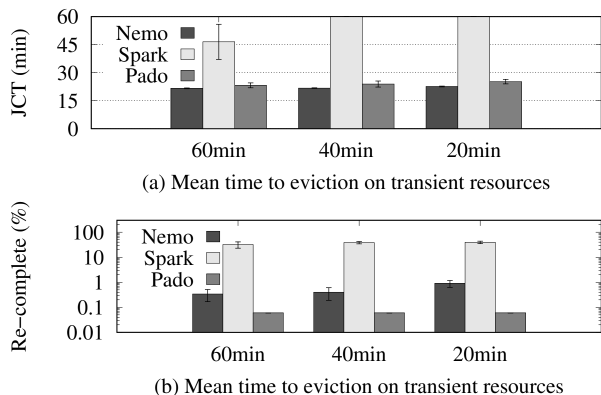


Figure 7: JCT and ratio of re-completed tasks to original tasks for different mean times to eviction on transient resources.

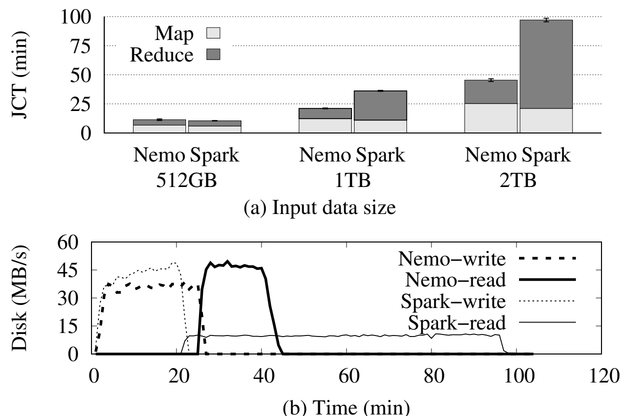


Figure 8: JCT for different input data sizes, and mean throughput of scratch disks for maintaining intermediate data when processing the 2TB input data.

Nemo optimized with the `GeoDistPass`, are shown on Figure 6 (a). Spark degrades significantly with larger bandwidth heterogeneity, since tasks that fetch data through slow network links become stragglers. In contrast, Iridium and Nemo are stable across different network speeds. Figure 6 (b) shows that the cumulative distributive function (CDF) of shuffle read time has a long tail for Spark compared to Iridium and Nemo. Iridium and Nemo show comparable performance with similar largest shuffle read blocked times, although Iridium shows overall better shuffle read blocked times using a more sophisticated linear programming model.

Transient Resources: Based on existing works [42,47,48], we classify resources that are safe from eviction as reserved resources and those prone to eviction as transient resources. We set up 10 EC2 instances for providing transient resources and 2 instances for reserved resources. When an executor running on transient resources is evicted, we allow the system to immediately re-launch a new executor using the transient resources to replace the evicted executor as described in Pado [48]. To evaluate handling long and complex DAGs with transient resources, we run an Alternating Least Squares [23] (ALS) workload, an iterative machine learning recommendation algorithm, on 10GB Yahoo! Music user ratings data [10] with over 717M ratings of 136K songs given by 1.8M users. We use 50 ranks and 15 iterations for the parameters. By varying the mean time to eviction for transient resources, we show how systems deal with the different eviction frequencies. The distribution of the time to eviction is approximated as an exponential distribution, similar to TR-Spark [47].

Figure 7 (a) shows the JCT of Pado, Spark and Nemo optimized with the `TransientResourcePass` for different mean times to eviction. With the 40-minute and 20-minute mean time to eviction, Spark is unable to complete the job even after running for an hour, at which point we stop the job. The main reason is heavy recomputation of intermediate data across multiple iterations of the ALS algorithm, which is repeatedly lost in recurring evictions. On the other hand, Nemo and Pado successfully finish the job in around 20 minutes, as both systems are optimized to retain a set of selected intermediate data on reserved resources. Figure 7 (b) shows the ratio of re-completed tasks to original tasks for different mean times to eviction. It shows that Nemo and Pado re-complete significantly fewer tasks compared to Spark, leading to a much shorter JCT. Nemo and Pado show comparable performance although Nemo re-completes more tasks, because the tasks that both systems re-complete are executed quickly and do not cause cascading recomputations of parent tasks.

Large-Shuffle Data: We evaluate how Nemo and Spark handle large shuffle operations using 512GB, 1TB, and 2TB data of the Wikimedia pageview statistics [7] from 2014 to 2016, as the datasets provide sufficiently large amount of real-world data. We use a Map-Reduce application that computes the sum of pageviews for each Wikimedia project. We choose the ratio of map to reduce tasks to 5:1, similar to the ratios used in Riffle [51] and Sailfish [35], and use 20 EC2 instances to run the workload.

The JCT of Spark and Nemo optimized with the `LargeShufflePass` are shown on Figure 8 (a). Both show comparable performance for the 512GB dataset, but Nemo outperforms Spark with larger datasets. To understand the difference, we measured the mean throughput of the disks used for intermediate data. Figure 8 (b) illustrates the mean disk throughput of scratch disks used for intermediate data when running the 2TB workload. Here, a spike in the write throughput is followed by a spike in the read throughput, which illustrates disk writes during the map stage followed by disk reads during the reduce stage while performing the shuffle operation. For Spark, the disk read throughput during the reduce stage is as low as about 10 MB/s, indicating severe disk seek overheads. In contrast, the throughput is as high as 45 MB/s for Nemo, as the `LargeShufflePass` enables sequential read of intermediate data by the following reduce

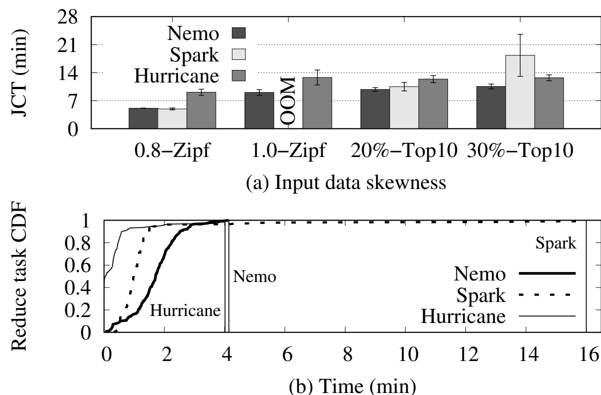


Figure 9: JCT for different input data skewness, and CDF of reduce task completion time when processing the 30%-Top10 skewed data. Each vertical line in the CDF graph denotes the completion time of the slowest reduce task.

tasks, which minimizes the disk seek overhead.

To measure the overhead of the `Relay` vertex inserted by the `LargeShufflePass` before the reduce operation, we have also run the 2TB workload on Nemo without the `LargeShufflePass`. The reduce operation begins 56 seconds earlier without the `LargeShufflePass` and the `Relay` vertex, where 56 seconds represent 2.05% of the JCT of Nemo with the `LargeShufflePass`.

Skewed Data: To experiment with different degrees of data skewness, we generate synthetic 200GB key-value datasets with two different key distributions: Zipf and Top10. For the Zipf distribution, we use parameters 0.8 and 1.0 with 1 million keys [12]. Datasets with Top10 distribution have heaviest 10 keys that represent 20% and 30% of the total data size. We run a Map-Reduce application that computes the median of the values per key on 10 EC2 instances. Because this application is non commutative-associative, for evaluating Hurricane we use an approximation algorithm similar to Remedian [39] to fully leverage its task cloning optimizations [12]. The Hurricane application also uses 4MB data chunks and uses its own storage to handle input and output data, similar to the available example application code.

Figure 9 (a) shows the JCT of Hurricane, Spark, and Nemo optimized with the `SkewCTPass` and the `SkewRTPass`. Performance of Spark degrades significantly with increasing skewness. Especially, Spark fails to complete the job with the 1.0 Zipf parameter, due to the load imbalance in reduce tasks with skewed keys which leads to out-of-memory errors. In contrast, both Nemo and Hurricane handle data skew gracefully. In particular, Nemo achieves high performance, and at the same computes medians correctly without using an approximation algorithm.

Figure 9 (b) shows the CDF of reduce task completion time when processing the 30%-Top10 dataset. The CDF for Spark shows that reduce tasks with popular keys take a significant amount of time to finish compared to other tasks. In contrast, the slowest task completes much quicker for Hurricane and Nemo. We have observed short-lived tasks alongside with longer tasks in Hurricane with its task cloning optimization, and longer tasks with balanced completion times for Nemo with its data repartitioning optimization.

Skewed data on Geo-distributed	Large Shuffle on Transient	Large Shuffle with Skewed
DP: OOM	DP: 100m	DP: OOM
GDP: OOM	TP: OOM	LSP: OOM
SKP: 27.2m	LSP: 100m	SSP: OOM
GDP + SKP: 14.9m	TP + LSP: 48.2m	LSP + SSP: 31.4m

Table 2: JCT when using different combinations of `DefaultPass` (DP), `GeoDistPass` (GDP), `SkewCTPass` (SKP), `TransientResourcePass` (TP), `LargeShufflePass` (LSP), and `SkewSamplingPass` (SSP).

To measure the overhead of the `Trigger` vertex inserted by the `SkewCTPass`, we also run the 30%-Top10 workload on Nemo without the `SkewCTPass` and the `SkewRTPass`. The reduce operation begins 35 seconds earlier without the `Trigger` vertex, where 35 seconds represent 5.52% of the JCT of Nemo configured with the `SkewCTPass` and the `SkewRTPass`.

These results for each deployment scenario show that each optimization pass on Nemo brings performance improvements on par with specialized runtimes tailored for the specific scenario.

5.2 Composability

We now evaluate combinations of different optimization passes. Table 2 summarizes the results.

Skewed Data on Geo-distributed Resources: In this experiment, we use the same 1.0-Zipf workload for the skew handling experiment in Section 5.1, because the workload showed the largest load imbalance. We use 10 EC2 instances representing geo-distributed sites with heterogeneous network speed in between 25Mbps to 2Gbps. Here, DP and GDP run into out-of-memory errors due to the reduce tasks with skewed keys that are requested to process excessively large portions of data. SKP and GDP+SKP both successfully complete the job with the skew handling technique in SKP, but GDP+SKP outperforms SKP by also benefiting from the scheduling optimizations in GDP.

Large Shuffle on Transient Resources: For this experiment, we use the same 1TB workload for the large shuffle experiment in Section 5.1, to use sufficiently large data that incurs disk seek overheads. In this case, we use 10 reserved instances and 10 transient instances with the 20-minute mean time to eviction setting.

Most notably, DP and LSP fail to complete even after 100

minutes, at which point we stop the job, and TP runs into out-of-memory errors. We have observed that heavy recomputation caused by frequent resource eviction significantly slows down the DP and LSP cases. We have also found out that the LSP optimization makes the application much more vulnerable to resource evictions compared to DP. The main reason is that with LSP, eviction of a single receiving task in the shuffle boundary leads to the entire recomputation of the sending tasks of the shuffle operation, to completely re-shuffle the intermediate data in memory. In contrast, DP does not need to recompute shuffle sending tasks whose output data are not evicted and stored in local disks. TP by itself also is not sufficient, as it leads to out-of-memory errors while pushing large shuffle data in memory from transient resources to reserved resources.

TP+LSP is the only case that successfully completes the job by leveraging both optimizations in TP and LSP. With TP+LSP, the job pushes the shuffle data from transient to reserved resources, and also streams them to local disks on reserved resources that are safe from evictions. This allows TP+LSP to handle frequent evictions on transient resources, and also to utilize disks for storing large shuffle data with minimum disk seek overheads. However, TP+LSP incurs the overhead of using only half of the resources (transient or reserved) for each end of the data shuffle. As a result, the JCT for TP+LSP with transient resources is around twice the JCT for LSP without using transient resources, which is displayed in Section 5.1. Nevertheless, we believe that this overhead is worthwhile, taking into account that transient resources are much cheaper than reserved resources from the perspective of datacenter utilization [38, 48].

Large Shuffle with Skewed Data: For this experiment, we generate a synthetic key-value dataset with a skewed key distribution that is around 1TB in size, as the datasets used in Section 5.1 for skew handling are not sufficiently large to incur disk seek overheads. This dataset has the distribution where heaviest 20 keys represent 30% of the total data size. Using this dataset, we run the same application that we have used for the skewed data experiment in Section 5.1 on 20 EC2 instances.

In this experiment, only SSP+LSP successfully completes the job, whereas all other cases run into out-of-memory errors. DP and LSP fails to complete the job, due to particular tasks assigned with excessively large portions of data, incurring out-of-memory errors. SSP by itself also runs into out-of-memory errors although it repartitions data across the receiving tasks of the shuffle boundary. We have observed that with large data size, the absolute size of the heaviest keys is significantly larger compared to smaller scale experiments with skewed data shown in Section 5.1. Without the LSP optimization, this problem is combined with random disk read overheads that degrade the running time of the shuffle receiving tasks, leading to out-of-memory errors. In contrast, SSP+LSP successfully completes the job by leveraging both of the optimizations from SSP and LSP.

These various results confirm that Nemo can apply combinations of distinct optimization passes to further improve performance for deployment scenarios with a combination of different resource and data characteristics.

5.3 Reusability

Finally, we evaluate how the same Nemo policy optimizes different applications. In addition to different applications used in prior experiments, we apply the policies on several ad-hoc BeamSQL [1] TPC-H [9] queries (Q) with different scale factors (SF), as they are widely used for benchmarking distributed data processing systems. Here, 1 SF is approximately 1GB of input data. We specifically use workloads that handle smaller input and intermediate data compared to the previous experiments, and thus are much less affected by the issues that occur in the specific scenarios like disk-seek overheads and resource evictions.

First, using 20 nodes with the `LargeShufflePass`, we observe 20.8 minute JCT for SF1000 Q3 that is 25% smaller than the JCT without the optimization, but no significant performance improvements for SF1000 Q14. We also observe 41.1 minute JCT for SF3000 Q12 that brings 22% performance improvements. Second, we do not observe meaningful performance improvements for SF100 Q4 and Q13 with the `SkewCTPass` on 10 nodes, as the dataset is not skewed. Finally, using 8 transient nodes with the 10-minute expected eviction rate and 2 reserved resources, we apply the combination of the `TransientResourcePass` and the `LargeShufflePass` on SF100 Q4 and Q14. For the respective queries, we observe JCTs of 8.2 minutes and 3.4 minutes, which are smaller than when not applying the optimizations by 9% and 15%.

These results as well as the results of different workloads in previous experiments confirm that the same optimization passes on Nemo can speed up different workloads instantly, with varying degrees of effectiveness.

6 Related Work

Nemo builds on many years of research in dataflow processing, relational database, and compiler optimizations. Nevertheless, we believe the set of trade-offs we have chosen to design the IR DAG, optimization passes, and runtime extensions for optimizing distributed dataflow processing makes Nemo a unique system.

Dataflow processing: Nemo differentiates itself from the existing application-level [22] and runtime-level [4, 20, 22, 40] approaches to dataflow scheduling and communication optimizations by taking a middle ground approach. Nemo provides a policy interface that transforms an intermediate representation (IR) of applications to express indirect but fine control over distributed scheduling and communication.

Our decoupled system design and our DAG-based IR are similar to Musketeer [15]. However, our work is complementary to Musketeer, as we focus on providing fine control over physical scheduling and communication in our IR, whereas Musketeer focuses on dynamically mapping its IR to a range of different execution runtimes.

The SparkSQL Catalyst optimizer [11] takes as input a SparkSQL application and outputs a Spark RDD application, which Nemo can take as input. Compared to Nemo, Catalyst has more information about application semantics (e.g., 'Add' '1' and '2'), but has less fine control over scheduling and communication (e.g., speculative task cloning).

Recently proposed dynamic query optimizers [28, 29] for distributed dataflow processing runtimes operate on high-level logical plans for SQL queries. Leveraging the semantics of SQL queries and the runtime information, these optimizers focus on choosing an optimal logical plan, for example by finding an optimal join order. Nemo operates on a lower-level IR DAG that supports general dataflow processing applications, and provides the methods to configure scheduling and communication methods of each data-parallel operation in the applications.

Weld [32] takes as input code that composes imperative libraries such as Pandas [30] and Numpy [41], creates a combined Weld IR program, and outputs optimized assembly code using LLVM. Weld can reduce data movement overheads across such imperative libraries, but it is not designed to optimize distributed scheduling and communication like Nemo.

Relational databases: Many of the optimizations in Nemo, such as parallelization and distributed scheduling optimizations, can be traced to research in parallel databases [14, 16]. Nemo enables expressing and composing various types of such optimizations for distributed dataflow processing applications, by introducing a policy interface that provides fine control and at the same time ensures correctness.

Our idea of annotating operators with execution properties is similar to using query hints in relational databases to influence the optimizer [13]. Nevertheless, these works focus on restricting the search space of SQL query execution plans, whereas Nemo focuses on tuning the scheduling and communication of dataflow processing applications.

Compilers: Our approach of expressing optimizations as passes that transform an IR is similar to LLVM [26]. However, in contrast to the LLVM IR that represents assembly code, the Nemo IR explicitly captures the dependencies and the communication patterns of coarse-grained, data-parallel operations. This enables passes on Nemo to express various distributed scheduling and communication optimizations.

Verified compilers, such as CompCert [27], aim to ensure the correctness of optimized assembly code using formal verification methods. Nemo aims to ensure the correctness of optimized distributed execution of dataflow processing applications, by introducing utility vertices and execution properties that make it simple to ensure correctness.

7 Discussion

Nemo provides a programming interface for building correct, reusable, and composable optimization policies. We discuss several directions to extend the interface and further facilitate the development of new policies.

Ensuring resource constraints: Although Nemo provides execution properties to specify where to place computations and data, Nemo relies on the runtime to determine the actual resources to acquire. To ensure that the resource constraints are met in the execution, we can incorporate the information into the IR DAG on the resource availability and acquisition.

Declaring optimizations ahead of time: To enable compile-time analysis of run-time pass conflicts and optimizations, we can provide the option to declare intended optimizations ahead of time. For example, we can receive more explicit information on the predicates (e.g., is a shuffle edge) and actions (e.g., store in memory) that a run-time pass intends to use.

Leveraging historical information: We can enable passes to use information on previous executions of the same application, and employ more sophisticated techniques such as machine learning to determine how to transform the IR DAG. To facilitate this, we can maintain a database that stores the information of the executed IR DAGs and their performance metrics, and provide an interface for passes to access the information in the database.

8 Conclusion

We presented Nemo, an optimization framework that provides fine control over distributed scheduling and communication of data processing applications, and at the same time ensures correct application semantics. We hope Nemo serves as a platform for dataflow optimization research and development. Nemo is available at https://nemo.apache.org.

Acknowledgments

We thank our shepherd Ashvin Goel and the anonymous reviewers for their feedback. We thank Sunghwan Ihm, Brian Cho, Rodrigo Fonseca, and members of the Software Platform Lab at Seoul National University for their comments on the draft. We also thank Min Hyeok Kweun, Jaehyeon Park, Seonghyun Park, Woo-Yeon Lee, Taegeon Um, Gyewon Lee, Yunseong Lee, Eunji Jeong, and Soojeong Kim for helping to implement and evaluate Nemo. Finally, we are grateful to the Apache Nemo community for their contributions. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Apache Beam. https://beam.apache.org.

[2] Apache Hadoop. https://hadoop.apache.org.

[3] Apache Nemo. https://nemo.apache.org.

[4] Apache Spark. https://spark.apache.org.

[5] Dryad Research Prototype. https://github.com/MicrosoftResearch/Dryad.

[6] Linux Traffic Control. https://lartc.org/manpages/tc.txt.

[7] Page view statistics for Wikimedia projects. https://dumps.wikimedia.org/other/pagecounts-raw.

[8] The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml.

[9] TPC-H. http://www.tpc.org/tpch.

[10] Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0. https://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

[11] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *ACM SIGMOD*, 2015.

[12] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. 2018.

[13] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. Power hints for query optimization. In *ICDE*, 2009.

[14] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 1990.

[15] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: All for one, one for all in data processing systems. In *EuroSys*, 2015.

[16] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[17] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, 2012.

[18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[19] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI*, 2017.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[21] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 2011.

[22] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, 2013.

[23] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[24] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SOCC*, 2010.

[25] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD*, 2012.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[27] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 2009.

[28] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. Rios: Runtime integrated optimizer for spark. In *SOCC*, 2018.

[29] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using qoop. In *OSDI*, 2018.

[30] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, 2010.

[31] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[32] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[33] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *ACM SIGCOMM*, 2015.

[34] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *SOCC*, 2012.

[35] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *SOCC*, 2012.

[36] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *SOCC*, 2012.

[37] 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.

[38] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report. https://github.com/google/cluster-data.

[39] Peter J Rousseeuw and Gilbert W Bassett Jr. The remedian: A robust averaging method for large data sets. *Journal of the American Statistical Association*, 1990.

[40] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *ACM SIGMOD*, 2015.

[41] SciPy.org. NumPy. https://www.numpy.org.

[42] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *EuroSys*, 2016.

[43] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[44] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, 2016.

[45] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[46] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russel Sears, Beysim Sezgin, and Julia Wang. Reef: Retainable evaluator execution framework. In *ACM SIGMOD*, 2015.

[47] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *SOCC*, 2016.

[48] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *EuroSys*, 2017.

[49] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[51] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *EuroSys*, 2018.

[52] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.

Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics

Yuzhen Huang*, Xiao Yan*, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhan Liu, Shuo Tu
The Chinese University of Hong Kong

Abstract

Data analytics frameworks that adopt immutable data abstraction usually provide better support for failure recovery and straggler mitigation, while those that adopt mutable data abstraction are more efficient for iterative workloads thanks to their support for in-place state updates and asynchronous execution. Most existing frameworks adopt either one of the two data abstractions and do not enjoy the benefits of the other. In this paper, we propose a novel programming model named *MapUpdate*, which can determine whether a distributed dataset is mutable or immutable in an application. We show that MapUpdate not only offers good expressiveness, but also allows us to enjoy the benefits of both mutable and immutable abstractions. MapUpdate naturally supports iterative and asynchronous execution, and can use different recovery strategies adaptively according to failure scenarios. We implemented MapUpdate in a system, called Tangram, with novel system designs such as lightweight local task management, partition-based progress control, and context-aware failure recovery. Extensive experiments verified the benefits of Tangram on a variety of workloads including bulk processing, graph analytics, and iterative machine learning.

1 Introduction

Existing offline data analytics frameworks can be roughly classified into two categories according to their data abstractions: **immutable** or **mutable**. The choice of data mutability results in two sets of fundamentally different system features and complex trade-offs between *efficiency* and *robustness* [1].

MapReduce [17] and Spark [70] are representative systems that adopt immutable data abstractions, where data accesses are bulk data movements. MapReduce and Spark provide effective *straggler mitigation* (by speculative execution) and efficient *failure recovery* (by recomputing only the lost partitions), which are critical for large-scale production deployment. As their immutable data abstractions imply a *bulk synchronous parallel* (*BSP*) execution model and lack support for in-place update, MapReduce and Spark do not perform well for workloads that benefit from fine-grained state access and asynchronous execution [2], e.g., sparse logistic regression and single source shortest path (SSSP).

There are also many systems that adopt mutable data abstractions to accelerate iterative workloads, such as vertex-centric graph systems (e.g., Pregel [36], GraphLab [34, 35], PowerGraph [21]) and machine learning systems based on the parameter server architecture (e.g., Parameter Server [31], Petuum [60], TensorFlow [1]). Mutable abstractions enable features such as fine-grained state access and asynchronous execution, which result in enhanced performance for iterative graph analytics and machine learning workloads. However, mutable abstractions make failure recovery and straggler mitigation more challenging. In these systems, failure recovery usually relies on a full restart from the latest checkpoint, and straggler mitigation with speculative execution is not supported as the backup tasks will conduct repetitive updates.

In general, immutable data abstraction leads to efficient failure recovery and effective straggler mitigation, while mutable data abstraction supports a richer set of features at the expense of weaker robustness. We discuss in greater details the interplay among data abstractions, system features and robustness in Section 2. In summary, a clear distinction in existing systems is that they support either mutable or immutable data abstraction, and lack a mechanism to choose which abstraction to use according to a given workload. Our analysis on the trade-offs between the two data abstractions leads to the following questions: *Can we model both mutable and immutable data abstractions under a unified framework? Can the system determine which abstraction to use according to the workloads? Is it possible to provide efficient failure recovery and straggler mitigation under mutable abstraction?*

*Co-first-authors ordered alphabetically.

[1] We refer robustness to efficient failure recovery and straggler mitigation.

[2] Compared with BSP, asynchronous execution by *stale synchronous parallel* (*SSP*) [24, 60] or *asynchronous parallel* (*ASP*) [49, 58] allows the machines/objects to have different progresses in iterative applications. The progress differences among the machines are bounded in SSP but unbounded in ASP.

We propose a distributed data analytics system, called **Tangram**, to bridge the gap between mutable and immutable data abstractions. Tangram adopts a new programming model, **MapUpdate**, which is used in the form ***A.map(B).update(C)***, where *A*, *B* and *C* are distributed data collections. Similar to MapReduce, the map tasks in MapUpdate are coarse-grained and side-effect-free, which allows speculative execution for straggler mitigation and failure recovery by only recomputing the lost partitions. However, the side-input (i.e., collection *B*) is read on a per-record basis and the update tasks conduct in-place update on states (i.e., collection *C*) to make recent updates visible to the map tasks. On this basis, MapUpdate also inherently supports iterative and asynchronous execution.

MapUpdate provides a simple rule to determine whether a collection should be mutable (Section 3), thus enabling the system to use mutable or immutable data abstraction for each collection adaptively according to a given workload. With this adaptability, Tangram provides elegant implementations for workloads including *bulk processing*, *vertex-centric graph analytics* and *iterative machine learning* (Section 4). In addition to good expressiveness, the ability to determine data mutability also enables the system to apply different failure recovery strategies for immutable and mutable collections, providing similar robustness as immutable systems (Section 5).

Tangram translates a MapUpdate plan (i.e., an invocation of "*A.map(B).update(C)*") into a number of *map* and *update* tasks. To reduce the overhead of centralized scheduling, Tangram uses a lightweight local task management strategy to schedule the execution of the tasks and to resolve access conflicts on each machine. A partition-based progress control mechanism is introduced to support iterations and asynchronous execution (Section 5). To achieve high efficiency, Tangram also incorporates optimizations such as delay combiner, process cache, and local zero-copy communication.

Our experiments show that Tangram provides efficient failure recovery and effective straggler mitigation. We also implemented a variety of workloads (e.g., bulk processing, machine learning, graph analytics, distributed crawler) on Tangram and compared their performance with specialized systems and highly optimized low-level MPI implementations. We found that Tangram can concisely express these workloads in an intuitive manner and the experiments show that Tangram's performance is comparable with that of specialized systems. Tangram's expressiveness and efficiency are especially useful for pipelined workloads consisting of multiple types of tasks as it eliminates context switch overheads.

Our main contributions can be summarized as follows:

- An in-depth analysis of the interplay among data abstraction, system features and robustness in existing systems. (Section 2)
- A novel programming model that can determine data mutability and can model both mutable and immutable data abstractions, resulting in good expressiveness. (Sections 3 and 4)
- A set of novel designs (e.g., partition-based progress control) and optimizations (e.g., delay combiner) that support different workloads efficiently on a common runtime. (Section 5)
- A comprehensive evaluation of Tangram's performance on a variety of workloads. (Section 6)

2 Immutable and Mutable Abstractions

We review related systems and analyze the complex interplay among their data abstractions, key features, failure recovery and straggler mitigation strategies. For convenience of discussion, we refer to systems that adopt an immutable/mutable data abstraction as **immutable/mutable systems**. We give a summary in Table 1 and discuss the details below.

Data-parallel analytics frameworks such as MapReduce [17], DryadLINQ [68] and Spark [70] are typical examples of immutable systems. They use functional dataflow graphs to model the dependency among datasets and break a job into multiple stages with dependency. The parallel tasks in each stage are independent and the stages are executed in a synchronous manner in which a stage can only start after its predecessors finish. This execution model enables straggler mitigation with speculative execution, which has been widely adopted and optimized in practice [3–5, 17, 71].

Immutable systems also provide efficient lineage-based failure recovery, for which only the lost data partitions are reconstructed from their parent partitions in the lineage graph. Compared with checkpoint-based recovery, lineage-based recovery can distinguish failure scenarios and does not need to roll back to the latest checkpoint upon every failure. For example, in K-means, if a machine holding a part of the training samples fails (i.e., narrow dependency), Spark only needs to reload the lost samples in parallel and recompute their updates to the centers. Only in cases such as PageRank, when the rank values of some vertices are lost (i.e., wide dependency), a full re-computation from the latest checkpoint is required. Moreover, Spark only checkpoints/reloads datasets that have a long lineage graph containing wide dependency (e.g., the rank RDD in the above example), which is more efficient than checkpointing all RDDs involved in computation.

Immutable systems are inherently stateless and only support BSP. However, many iterative workloads have intuitive stateful representations (e.g., the rank values in PageRank, and the model parameters in sparse logistics regression) and can benefit from asynchronous execution. For example, machine learning algorithms such as stochastic gradient descent (SGD) converge faster under SSP and ASP [24, 49, 69], and it has also been proven that a number of asynchronous graph algorithms have faster convergence compared to their synchronous counterparts [49, 73]. Therefore, many specialized mutable systems such as vertex-centric graph systems [21, 35, 36] and parameter-server-based machine learning systems [31, 60] support in-place state updates and asynchronous execution.

Table 1: The data abstractions and key features of some representative systems

Category	Systems	Usability	Abstraction		System Support		
		Programming Model	State Representation	Access Pattern (Shuffle)	Execution Model	Straggler Handling	Failure Recovery
Stateless Dataflow	MapReduce [17], DryadLINQ [68], Spark [70]	functional	immutable	coarse-grained	BSP	speculative execution	lineage and checkpoint
Distr. Shared Mem.	Piccolo [46]	push/pull	mutable kv table	fine-grained	BSP	task stealing	checkpoint
Parameter Server based	Parameter Server [31]	push/pull	mutable kv table	fine-grained	BSP/SSP/ASP	N/A	replication
	Petuum [60]	push/pull	mutable kv table	fine-grained	BSP/SSP/ASP	N/A	checkpoint
Distributed Graph	Pregel [36]	vertex-program	mutable state	coarse-grained	BSP	N/A	checkpoint or message replay
	GraphLab [34, 35], PowerGraph [21]	vertex-program	mutable state	fine-grained	BSP/ASP	N/A	checkpoint
	Tangram	functional map in-place update	immutable/mutable	partition-based	BSP/SSP/ASP	partition migration	lineage and checkpoint

Asynchronous execution also makes mutable systems more robust to micro stragglers [3] as fewer barriers are enforced. In contrast, immutable systems are prone to micro stragglers as their BSP execution model enforces a synchronization barrier in every iteration.

However, mutable systems only provide sub-optimal system-level solutions to straggler mitigation and failure recovery. Most of the mutable systems in Table 1 rely on the nature of the applications for straggler mitigation and do not provide a system-level support. For example, graph systems such as Pregel [36] and PowerGraph [21] rely on graph partitioning to ensure a balanced workload distribution among workers. Parameter Server [31], Petuum [60], GRACE [58] and Maiter [73] utilize the asynchronous nature of the application algorithms to mitigate micro stragglers. Mutable systems do not support speculative execution as updates are conducted in a fine-grained manner and it is costly to keep track of the committed writes in order to avoid repetitive updates. Instead, mutable systems typically use task stealing to handle stragglers [23, 46].

For fault tolerance, mutable systems usually require a full restart from the latest checkpoint (e.g., Petuum, Pregel), or use expensive replication when recovery time is critical (e.g., Parameter Server). Contrary to immutable systems, mutable systems often recompute everything from the latest checkpoint. In addition, any failure would cause these systems to discard and reload all data. The key problem is that these systems do not distinguish the mutable and immutable parts in an application. Although existing mutable systems can be modified individually to support more efficient fault tolerance, we offer a unified mechanism to solve this problem, which is especially useful for pipelined workloads where datasets can change between mutable and immutable status (e.g., the TF-IDF vectors in the pipelined workload in Section 6.2).

[3] *Micro stragglers* are transiently stalling workers and may be caused by packet loss, system cron jobs, etc. *Macro stragglers* are slow due to more persistent reasons, such as workload imbalance and resource contention.

3 Programming Model

In this section, we introduce our MapUpdate programming model and discuss its differences from MapReduce and stream processing frameworks.

3.1 MapUpdate

The basic data abstraction in our MapUpdate programming model is *collection*, which contains a set of objects (or records) and is usually kept in memory. A collection is divided into partitions and partitions are distributed across machines in the cluster (with hash partitioner by default, but configurable by users). The **MapUpdate** programming model is typically used in the form of

A.map(B, map_func).update(C, update_func),

in which A, B, and C are *map collection*, *side-input collection*, and *update collection*, respectively. The *map_func* has a signature $(T, [S, \ldots]) => seq(K, V)$, which takes in an object of type T in the map collection and (optionally) handler(s) S to the side-input collection(s), and generates some key/value (K/V) pairs. The *update_func* has a signature $(U*, V) => nil$, which takes in a pointer $U*$ to an object in the update collection and an update value V, and returns nothing.

To execute a MapUpdate command (also called a *plan*), a machine launches parallel map tasks on its local partitions of the map collection, where each map task performs *map_func* on the objects in one partition of A to generate intermediate results. The *map_func* may use the information (parameters) provided in the side-input collection B. The intermediate results (of a map task) are then shuffled according to their keys and committed to the corresponding objects in update collection C with the *update_func*. By default, the retrieval of the objects in the side-input collection is fine-grained (i.e., on a per-key basis), while the shuffle of intermediate results and modification to the update collection are conducted on a per-partition basis. MapUpdate associates progress with each partition and allows different partitions to have differ-

ent progresses, and state access is also progress dependent (Section 5.2). This *partition-based state access pattern* of MapUpdate is different from the *coarse-grained state access pattern* in dataflow systems (e.g., Spark), in which all partitions have the same progress (i.e., BSP). Task execution in the Tangram system is also partition-based, i.e., a partition is the granularity of task execution.

MapUpdate has an explicit side-input collection. In contrast, without the side-input, existing systems (e.g., Spark) may use broadcast for state sharing, which is inefficient for large and sparse states (e.g., sparse logistic regression). Some other systems (e.g., Google Dataflow [2]) also support the side-input collection but require it to be small and immutable. MapUpdate does not have such constraints, enabling it to succinctly and efficiently express workloads that have intuitive stateful representations (see examples of machine learning and graph analytics applications in Section 4). MapUpdate also does not require A, B and C to be different collections. When $A = C$ or $B = C$, by default, MapUpdate does not make a copy of C for read. Instead, MapUpdate reads and writes the same collection, which enables the map tasks to see the latest (maybe inconsistent) updates. As we will show in Section 4, this is important for the asynchronous execution of workloads such as distributed crawler as they can tolerate inconsistent states and benefit from fewer synchronization barriers.

MapUpdate ensures consistency [4] if a plan (e.g., word count) does not write/read the same collection. For plans that write/read the same collection, consistency is not guaranteed. This is not problematic because applications such as SGD can trade consistency for efficiency without sacrificing correctness, and many specialized systems deliberately incorporate designs to benefit from that. When strict consistency is required, users can create another copy of the read collection for write as in Spark and Piccolo. MapUpdate does not enforce any order when committing the updates of the map tasks, and the results of a plan may not be deterministic for some applications (e.g., SGD based logistic regression). In the face of failure, MapUpdate ensures that each update is committed exactly once.

In general, the map collection A contains the input data, such as samples in machine learning and documents in word count, while the side-input collection B provides information needed in computation, such as model parameters in machine learning. The update collection C holds the computation results, e.g., the final count values in word count. The side-input collection can be omitted, for example, a user can write *docs.map(map_func).update(count, update_func)* for word count. Instead of a single collection and function, users can provide multiple side-input collections, update collections, and update functions. Users can easily specify how a plan is executed using the configurations in Table 2, for example, *A.map(B).update(C).setIter(100).setStaleness(2)* will

[4] At iteration t, a read on data sees all updates from iteration smaller than t but not updates from iteration equal or larger than t.

Table 2: Configurations in MapUpdate

Configuration	Description
setIter(*int n*)	Run for *n* iterations
setStaleness(*int s*)	Set staleness to *s*
setCombine(*func*)	Register combiner
setCheckpointInterval(*int n*)	Set checkpoint interval

conduct the MapUpdate plan for 100 iterations using SSP with $staleness = 2$. Additionally, *setCombine(func)* provides a function to combine the map outputs before shuffling for communication reduction, and *setChecktpointInterval(n)* configures the checkpoint interval in an iterative application. We will show how these flexibilities of MapUpdate translate into good expressiveness in Section 4.

3.2 Comparison with Existing Frameworks

We highlight the main differences between MapUpdate and MapReduce [17], Flink [10] and Spark Structured Streaming [6] in this section.

MapReduce. MapUpdate differs from MapReduce in several important aspects. First, while *map* in MapUpdate is functional (similar to *map* in MapReduce), *update* allows for asynchronous in-place modification to the stateful collection. Second, MapUpdate allows the side-input collections to be specified explicitly and access to the side-input collections is fine-grained, which improves efficiency in many applications such as machine learning and graph analytics. Third, with designs to be introduced in Section 5, MapUpdate provides support for iteration and consistency protocols including BSP, SSP and ASP (configurable by *setStaleness*). Although there are attempts to support key-value-style update (IndexedRDD [26]) and iteration (HaLoop [9], iterative MapReduce [18], Map-Reduce-Update [8]) under the MapReduce framework, MapUpdate is fundamentally different as these systems do not support in-place updates and asynchronous execution due to their data immutability.

The most important contribution of MapUpdate, however, is that it provides a simple mechanism for the system to determine whether a collection is mutable in a plan from the API call: *the update collection is mutable, and other collections, if different from the update collection, are considered immutable*. For example, a MapUpdate plan training a logistic regression model may be expressed as *samples.map(param).update(param)*, and the system can infer that collection *param* (storing the parameters) is mutable and collection *samples* (storing data samples) is immutable. The ability to determine data mutability allows the system to distinguish failure scenarios and provides efficient failure recovery strategies accordingly as in immutable systems. For example, when a machine fails, the system can determine whether the failed machine holds partitions of *param*. If not, only the lost partitions of *samples* need to be reloaded. Otherwise, the system rolls back to the latest checkpoint for *param*, but the partitions of *samples* on the healthy machines do not

need to be reloaded.

To support speculative execution, MapUpdate restricts updates to be conducted on a per-partition basis, in which updates from a map partition are committed together. This per-partition update strategy enables Tangram to record which partition has already committed update and is crucial for speculative execution.

Stream Processing Frameworks. Modern stream processing systems such as Flink and Spark Structured Streaming also support both mutable and immutable abstractions but with restricted applicability. We discuss how MapUpdate is different from them here.

First, states in MapUpdate are shared and can be accessed globally, which allows Tangram to support workloads such as machine learning and graph analytics more efficiently. In contrast, states in Flink are bounded with operators and states in Spark Structured Streaming are restricted to key groups. In fact, states in Flink and Spark Structured Streaming are mainly designed for maintaining states across streaming records (e.g., for session tracking). Thus, they are not efficient for read/write in machine learning and graph analytics workloads, which introduce loops in the computation graph. Specifically, using loops in Flink requires to limit the input rate of the input stream to avoid deadlocks caused by cyclic backpressure [20], while Spark Structured Streaming does not allow loops in the dataflow graph, which is necessary when using the stateful operators for iterative workloads. In contrast, MapUpdate naturally supports iteration and in-place update.

Second, checkpointing in Flink and Spark Structured Streaming is more complicated (as they are designed for stream processing), while Tangram is designed for batch processing and only checkpoints mutable collections. Spark Structured Streaming uses checkpointing and write-ahead logs for fault tolerance. Flink also needs to restart from the latest checkpoint for any failure.

4 Applications

Programming with MapUpdate to construct data-parallel applications is simple: users define the collections, construct the MapUpdate plan by providing the *map/update* functions and specify the plan configurations. Low-level system issues such as parallelism and fault tolerance are hidden from users. In this section, we demonstrate how MapUpdate can be used to implement a wide range of applications.

4.1 Bulk Processing

MapUpdate can easily implement the bulk processing workloads targeted by MapReduce, which are usually stateless, non-iterative and involve only bulk data movement. We illustrate by the word count example, which is similar to the one in Spark: the map function generates (word, count) pairs when scanning local documents, while the update function aggregates the (word, count) pairs for final counts. Note that in the plan of word count, there is no side-input collection.

```
// Doc: (word1, word2...): (string, string...)
// WordCount: (word, count): (string, int)
// docs: collection<Doc>
// wordcount: collection<WordCount>
docs.map(doc => (w, 1) for each word w in doc)
    .update(wordcount, (wc, c) => wc.count += c)
```

4.2 Iterative Machine Learning

Iterative machine learning algorithms repeatedly refine a set of model parameters with updates computed from the training samples. These algorithms (e.g., SGD) are usually robust to asynchronous execution, in which update is calculated using outdated or inconsistent model parameters. Parameter-sever-based systems (e.g., Parameter Server [31], Petuum Bösen [60]) are widely used for distributed machine learning and support SSP and ASP to benefit from asynchronous execution. Tangram can model parameter server by using the model parameters as both the side-input collection and update collection. We show an example of training logistic regression using SGD with SSP ($s = 2$). The map function calculates the stochastic gradient of local samples using the model parameters, while the update function commits the gradient updates to the model parameters. Iteration and asynchronous execution can be configured using the *setIter* and *setStalenss* commands in Table 2. Note that when *setStalenss* is not configured, Tangram uses BSP by default.

```
// Sample: (label,(k,v)..): (int, (int,float)..)
// Param: (k,v): (int, float)
// data: collection<Sample>
// params: collection<Param>
map_func(Sample sample, Params params, Output o):
  grad = CalcGrad(sample, params)
  o <- grad  // grad: ((k,v)...)

update_func(Param param, float update):
  param.val -= learning_rate * update

data.map(params, map_func)
    .update(params, update_func)
    .setIter(100).setStalenss(2)
```

4.3 Vertex-Centric Graph Analytics

Vertex-centric graph analytics systems (e.g., Pregel [36], PowerGraph [21]) usually update vertex states iteratively according to the states of neighboring vertexes. Tangram can model vertex-centric graph processing by using the vertex state collection as both map collection and update collection [5]. We use PageRank as an example. The map function calculates the contribution of a vertex's PageRank value to its out-neighbors, while the update function merges the contributions from the in-neighbors. The ranks and the links collection are

[5]Using vertex state as side-input and update collection is also feasible.

co-partitioned (by using the same partitioner) to reduce communication overhead. Similarly, Tangram can also implement the edge-centric model [51].

```
// Rank: (id, pr): (int, float)
// Link: (id, nb1, nb2...): (int, int, int...)
map_func(Rank r, Links links, Output o):
  for each neighbor nb in links[r.id]:
    o <- (nb.id, 0.85 * r.pr/len(links[r.id]))

update_func(Rank r, float contrib):
  r.pr += contrib

ranks.map(links, map_func)
     .update(ranks, update_func)
     .setIter(30)
```

4.4 Distributed Crawler

Tangram supports crawler by using urls as both the map collection and update collection. The map function downloads the web page pointed by the current url and extracts new urls, while the update function inserts the new urls into the urls collection and marks the processed urls as visited. Note that there is no side-input collection. *setIter(-1)* keeps executing the iteration (i.e., keep crawling), while *setStaleness(-1)* means using ASP.

```
// Url: (url, status): (string, ToFetch/Done)
map_func(Url url, Output o):
  if url.status is ToFetch:
    new_urls = DownloadAndExtractNewUrls(url)
    for each new_url in new_urls:
      o <- (new_url, ToFetch)
    o <- (url, Done)

update_func(Url url, Status s):
  if url.status is not Done:
    url.status = s

urls.map(map_func)
    .update(urls, update_func)
    .setIter(-1).setStaleness(-1)
```

In the above applications, we use different combinations of the three collections (A,B,C) to achieve different computation patterns. Tangram also supports many other applications (e.g., Nomad [69] and graph matching [12]) that are hard to be implemented in existing systems.

4.5 Pipelined Workloads

MapUpdate is especially useful for pipelined workloads. In fact, the Tangram project was motivated by production data analytics workloads that are common in companies such as Alibaba, which consist of pipelines involving different types of tasks. Typical pipelines begin with MapReduce-style data processing, then conduct various advanced analytics (e.g., parameter-server-style model training), and end with testing and verification. We briefly describe a *user classification* pipeline and a *fraud detection* pipeline as examples.

In a user classification pipeline, users are divided into groups according to their purchase records to generate labels. The basic information (e.g., age, gender and location), search history and activity patterns (e.g., log-in frequency, active time period) are gathered from multiple tables using MapReduce-style join to produce features. Then, various machine learning models (e.g., logistic regression, SVM) are trained using a parameter-server-based framework to classify users into different purchase pattern groups. Lastly, the models are tested on a held-out dataset to select the best-performing one for use. We remark that user classification is only a component of the much larger item recommendation pipeline, which involves a more diverse set of workloads such as graph analytics and matrix factorization.

In a fraud detection pipeline, the goal is to find malicious sellers who use fake transactions to bump up their scales records [47]. The static relationship among users (i.e., buyers or sellers) and the dynamic payment activities are first processed, and a graph is extracted from the pre-processed data to model the buyer-seller interaction. Then, graph matching is applied to find interaction patterns that match some predefined templates corresponding to fraud patterns. Finally, these interactions are verified by further analysis and the results are used to update the fraud template library. The verification process typically involves MapReduce (e.g., joins to obtain details of suspected users) and graph analytics such as computing the distances from suspected users to blacklisted users.

As we will show in the experiments, processing different tasks in a pipeline with respective specialized systems introduces expensive context switch overheads for dumping/loading output/input data by the systems. Using many systems for a single pipeline also hurts robustness because different systems provide different fault tolerance semantics and require engineers to learn/tune all the systems. With the expressive API of MapUpdate, unified fault tolerance semantics and high efficiency, Tangram (our system that implements MapUpdate) can handle the entire pipeline in a unified framework and thus completely remove the context switch overheads. Moreover, the unified MapUpdate API also significantly reduces development costs without users' need to learn many systems.

5 System Design

Designing a system to support the MapUpdate API is challenging in the following aspects: (1) As tasks in MapUpdate have complicated interactions and dependencies (e.g., read/write conflicts, requiring remote data transfer), a low-overhead task management and scheduling strategy is crucial for efficiency. (2) MapUpdate supports iterative plans and flexible consistency control, which requires a distributed progress control protocol that enables various execution models (i.e., BSP, SSP and ASP) under a unified framework. (3) To achieve efficient failure recovery, effective mechanisms are needed to distinguish failure scenarios and apply different recovery

strategies accordingly as analyzed in Section 2.

Tangram adopts a master-worker architecture. The master is responsible for DAG scheduling (coordinating the workers to execute runnable plans), progress tracking (managing progress and collecting execution statistics from workers for fault tolerance and straggler mitigation), and partition management (keeping track of the location of the partitions by maintaining the master copy of the partition map). The workers serve as the distributed in-memory storage for the partitions and each worker uses a local controller to manage local task execution. For scheduling, the master only issues control commands (start, update progress, migrate, recover, etc.) to workers and the local controller is responsible for scheduling its own tasks. The local controller also synchronizes the local copy of the partition map and the execution progress with the master. This design reduces centralized scheduling overhead and is crucial for scaling to large clusters.

5.1 Local Task Management

The local controller in each machine manages three kinds of tasks, i.e., map task, respond task, and update task. A map task runs the user-defined map function for every object in a local map partition, combines the intermediate results locally if a combine function is provided, serializes the (combined) results and adds the results to the sender, which will send them to remote machines (according to the partition map) for update. A map task invokes a fetcher if it needs to fetch some records (e.g., parameters in machine learning) in the side-input collection. The requested records are indexed by keys and the fetcher splits these keys into multiple subsets, each corresponding to a partition of the side-input collection. Then the fetcher sends out the fetch requests to the remote controllers holding the records and blocks the map task. The fetch request will invoke a respond task in the remote controller and the map task will be unblocked when all the responses are received. An update task updates a local partition with the received intermediate results (from a map task) using the update function, while a respond task answers a fetch request using a local partition of the side-input collection.

Different from the pull-based shuffle mechanism in MapReduce-like systems (where reducers pull intermediate results from mappers), a push-based shuffle mechanism is used in Tangram, in which updates are pushed to the update partitions on a per-partition basis. Push-based shuffle can overlap network communication with the computation of map tasks, but the system needs to handle more complex read/write conflicts between tasks. To resolve the read/write conflicts for a partition, the controller enforces a simple access control strategy. It assumes that map and respond tasks read a partition, while update tasks write a partition. The controller ensures that writes to a partition are exclusive while reads are not. If there is an ongoing update task on a partition, then map tasks, respond tasks and other update tasks on the same partition will be blocked. If there is an ongoing map or respond task on a partition, then other map and respond tasks on this partition can still run but update tasks will be blocked.

The execution of a plan starts when the global scheduler instructs the local controller to push a number of map tasks to the map thread pool. When the controller receives a fetch/update request, it invokes a respond/update task. The respond/update task is pushed to the thread pool for execution if it satisfies the access control policy; otherwise, it will be inserted into a pending buffer. Once a task finishes, the controller will be notified and it will check the pending buffer to find tasks satisfying the access control policy and push them to the thread pool for execution. The local controller is implemented as a single-thread event loop and manages the pending buffer and all control-related data structures. The event-loop simplifies the implementation logic by avoiding complex locking.

5.2 Partition-Based Progress Control

As a partition is the granularity of task execution in Tangram, each partition can have its own progress. Therefore, Tangram uses a partition-based progress control mechanism to support the BSP, SSP and ASP execution models. Different from parameter-server-based systems, in which progress is associated with a worker, Tangram associates progress with a partition. The local controller records the map progresses of the local map partitions and the update progresses of the local update partitions. Note that when the map collection and the update collection are the same (e.g., in PageRank), a partition has both map progress and update progress. We will show that partition-based progress control also ensures correctness upon failure and improves recovery efficiency in Section 5.3.

At the start of a plan, the map progresses and the update progresses are initialized as zero. When a map task finishes, the map progress of the corresponding partition is incremented by one, and the update requests generated by this map task also carry the map progress (before increment). For an update partition, the local controller uses a bitmap (for each map progress) to record the map partitions for which update has already been committed. The controller sets the update progress of a partition as the minimum progress for which there are still missing updates. The controller assumes that a map partition will generate update for all partitions in the update collection and can confirm that all update requests are committed using the bitmap.

The controller sets its local progress as the minimum update progress of its partitions and reports it to the global scheduler. The global scheduler regards the minimum progress among workers as the global progress and broadcasts it to all workers upon changes. If the staleness is k and the global progress is m, the local controller will only schedule map tasks for its partitions with a progress no larger than $m+k$. An example of progress control is provided in Figure 1. Partition P1 in worker 0 has an update progress of 2 as it has not received

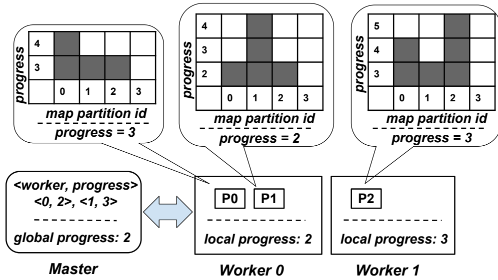


Figure 1: Example of progress management. Dark squares denote the received updates for each update collection.

the update with progress 2 from map partition 3. The local progress of worker 0 is 2 as the update progress of P0 and P1 are 3 and 2, respectively. Collecting the local progresses from worker 0 and worker 1, the master sets the global progress as 2. Once the update with progress 2 from map partition 3 is committed to P1, both the local progress of worker 0 and the global progress will be updated to 3.

Assume that there are M map partitions in total and a machine hosts n update partitions, the machine needs $O(MnT)$ memory to store the bitmap for a plan, where T is the number of active iterations (iterations for which there are uncommitted updates) and is usually small. We reduce the memory consumption of the bitmaps by deleting the bitmap for which all update requests have been committed and creating a bitmap only when receiving a new map progress. As the number of partitions is usually not large, the cost of progress control is acceptable as we will show in our experiments.

5.3 Context-Aware Failure Recovery

Tangram distinguishes two failure scenarios, i.e., *local failure* and *global failure*, and applies different recovery strategies.

Local failure is the case that the failed machines do not hold update partitions. Local failure does not directly affect the task execution on the healthy machines and is similar to losing RDDs with narrow dependency in Spark. In this case, Tangram only reloads the lost partitions on the healthy machines in parallel and sets their progresses as the current global progress. Some of the updates from the lost partitions may have been committed to the update collection and setting their progresses as the global progress may result in repetitive map tasks. Tangram rejects the repetitive updates generated by these map tasks using the bitmap.

Global failure happens when the failed machines contain partitions of the update collection. Examples include losing machines holding the model parameters in logistic regression or the rank values in PageRank. Global failure directly affects the computation on all machines and is similar to losing RDDs with wide dependency in Spark. In this case, Tangram reloads the lost mutable collection from the latest checkpoint and resets the global progress and the progresses of all partitions to the latest checkpoint. But for immutable collections, such as the graph links in PageRank, Tangram only reloads the lost partitions. The master assigns the tasks of loading the lost partitions to the healthy machines in a balanced manner so that the machines can recover from failure in parallel. Moreover, Tangram also respects the co-partitioning relation of the collections in failure recovery.

Tangram infers which collection is mutable and checkpoints only the mutable collections. Checkpointing is conducted in an asynchronous manner and on a per-partition basis, so that the execution of the entire plan does not need to be stopped. When the progress of an update partition reaches the checkpoint iteration, write access is blocked and a copy is written to disk along with the bitmap. Note that this checkpoint may be inconsistent, as the partition may have seen updated from iterations larger than the checkpoint iteration (under ASP or SSP). The bitmap is used to reject repetitive updates from these iterations during recovery. For pipelines that involve multiple plans, Tangram also checkpoints the mutable collections once a plan finishes so that failure recovery can be conducted inside a plan. Tangram only handles worker failures, while the master failure can be handled by a standby master (similar to the standby master in Spark [55]) but is not implemented in the current version.

5.4 Straggler Mitigation

Tangram uses partition migration for straggler handling. As update tasks and respond tasks are relatively lightweight, Tangram focuses on balancing the workload of map tasks by migrating map partitions. For non-iterative workloads (e.g., word count), the master monitors the number of ongoing and pending map tasks on each machine. If some machines do not have map tasks to run, the master migrates some of the map partitions from the heavily loaded machines to them. We allow the system to migrate map partitions with ongoing map tasks, which resembles speculative execution in Spark. Migration does not necessarily require data transfer from slow machines, as will be discussed later.

For iterative workloads, Tangram only handles macro stragglers, while micro stragglers can often be handled by asynchronous execution. By default (tunable by users), Tangram considers a machine as a macro straggler if its per-iteration time is more than 1.1 times of the median (of all machines) in three consecutive iterations or more than 1.5 times of the median in one iteration. Tangram also respects the co-partitioning relation among the collections to avoid high communication overhead after migration.

Tangram adopts different migration strategies for immutable and mutable partitions. For immutable partitions, the destination machines just load them from a shared storage system like HDFS rather than asking the source machine for transfer, since the source machine is already overloaded. For mutable partitions, Tangram uses a migration procedure similar to the two-stage migration in Piccolo [46].

The load of the system may be unbalanced due to skewed partition size (e.g., due to improper hash function). Currently, Tangram does not support online re-partitioning for workload redistribution. Similar to other systems, skewed partitions can be addressed by either fine-grained sharding (setting the number of partitions to be much larger than the number of machines) or providing a tailored partitioning function.

5.5 Communication Optimizations

Delay Combiner. In Tangram, combining the updates from many map tasks leads to higher compression ratio, but sending out the updates immediately reduces latency. We provide a delay combiner, in which users can specify the granularity of combining with a *combine_timeout*. Setting *combine_timeout* to 0 sends out the updates immediately, while setting *combine_timeout* to *kMaxCombineTimeout* combines all local map outputs in an iteration.

Process Cache. The process cache in Tangram is similar to the one in Petuum. Previously fetched records of the side-input collection and their versions are kept in the cache. A new fetch request will not be sent if the records with the required version are already in the cache.

Local Zero-Copy Communication. Tangram utilizes local zero-copy communication whenever possible: if an update request is to be sent to a local partition, it will be moved to the local controller and can be directly accessed by the update task. Similarly, zero-copy communication is also used for fetching local objects.

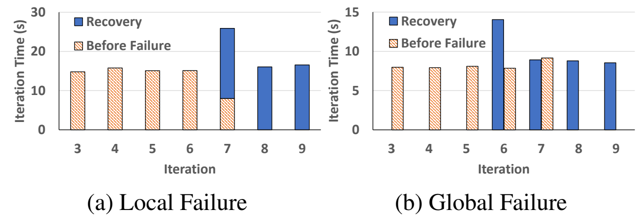
6 Experiments

We implemented Tangram in about 16K lines of C++ code. The communication module was built using ZeroMQ [72] and libhdfs3 [32] was used to exchange data with HDFS without the JNI overhead. Source code for the system and the applications in Section 4 can be found at https://github.com/Yuzhen11/tangram/. We evaluated Tangram on a cluster of 20 machines connected with 1 Gbps Ethernet. Each machine is equipped with two 2.0GHz E5-2620 Intel(R) Xeon(R) CPU (12 physical cores in total), 48GB RAM, a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), running on 64-bit CentOS release 7.2. We optimized the number of partitions for both Tangram and the systems we compared in the experiments.

6.1 Failure Recovery & Straggler Mitigation

Failure recovery and straggler mitigation are critical for data analytics in production. In this set of experiments, we show that Tangram achieves efficient failure recovery and effective straggler mitigation, even for workloads with mutable states, by distinguishing immutable and mutable collections.

Failure Recovery. We used two experiments to simulate different failure scenarios. In the first experiment, we unplugged



(a) Local Failure (b) Global Failure

Figure 2: Performance of failure recovery

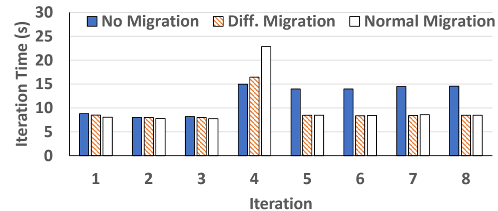


Figure 3: Performance of straggler handling

one machine holding the training data for K-means, which is a *local failure* according to Section 5.3. The second experiment unplugged one machine for PageRank, which corresponds to *global failure*.

We report the failure recovery performance under the two scenarios in Figure 2. For clearer presentation, we only plot the performance of Tangram, while we report the performance of Spark in text as a baseline. For local failure, Tangram took 17.8 seconds to reload the lost training data (∼6 GB) and finish the 7th iteration, while Spark took around 40 seconds to recover. Both Tangram and Spark did not restart the job from the latest checkpoint, but performed re-computation only on the lost partitions. In contrast, most of the mutable systems such as Petuum, PowerGraph and Naiad, would have to roll back to the latest checkpoint in case of any failure.

For the global failure (occurred at the 7th iteration), Tangram rolled back to the latest checkpoint (taken at the 5th iteration) and continued by re-executing the 6th iteration. The roll-back is necessary because some partitions of the (mutable) rank collection are lost. The longer recovery bar at the 6th iteration in Figure 2b includes the normal execution time for the iteration and another 5.2 seconds to reload the mutable collection (∼50 MB) and the lost immutable partitions (∼1 GB). In total, Tangram took 29 seconds to finish the 7th iteration, while Spark took 47 seconds. We note that Spark also requires a full re-computation from checkpoint in this case (i.e., long lineage with wide dependency [70]).

To provide a better picture of Tangram's failure recovery performance, we also implemented the baseline strategies (e.g., full reload, full checkpoint) in Tangram for a fair comparison. A full reload (as used in existing mutable systems) needs to load 121GB and 23GB data, and took 56.8 and 15.7 seconds (vs. Tangram's 17.8 and 5.2 seconds) for K-means and PageRank, respectively. In addition, distinguishing the immutable and mutable parts also results in more efficient checkpointing for PageRank, as checkpointing only the ranks

(∼1GB) took 5 seconds, while a full checkpoint (∼23GB) took 127 seconds. Tangram also supports asynchronous background checkpointing, which makes a copy of the mutable collection and writes the checkpoint in the background. Without background checkpointing, each checkpoint would take an extra 5 seconds for PageRank.

Straggler Mitigation. To test the performance of straggler mitigation, we used *cpulimit* tool to restrict one Tangram worker to have only 600% of the total 2400% cpu shares in the PageRank job at the 4th iteration, and the per-iteration time is reported in Figure 3. *Diff. migration* asks the straggler only for mutable partitions, while *Normal migration* asks the straggler for both mutable and immutable partitions. Both strategies were implemented in Tangram. The result shows that partition migration effectively reduces per-iteration time as an iteration took about 14.5 seconds without migration, but only 8.4 seconds with migration. In addition, distinguishing immutable and mutable collections also speeds up the migration. *Diff. migration* only requires the straggler to transfer the mutable partitions (ranks: ∼50MB) and reloads the immutable partitions (links: ∼ 1GB) from HDFS, which improves migration speed by approximately 38% (10 seconds vs. 16 seconds) compared with *Normal migration*.

6.2 Expressiveness and Efficiency

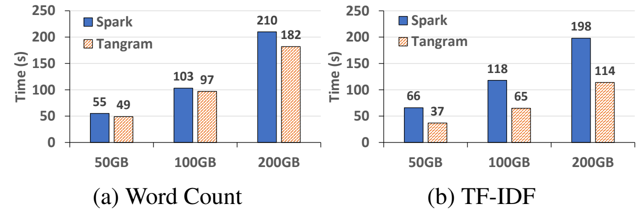
We have shown that the MapUpdate API is flexible and can express a wide variety of workloads in Section 4. In this set of experiments, we show that Tangram achieves comparable performance as specialized systems.

Bulk Processing. For non-iterative bulk processing workloads, e.g., MapReduce-style workloads, we tested word count and TF-IDF[6], and compared with Spark [70] (version 2.2.0). We replicated the Wikipedia corpus [19] to test the scalability of the systems and report the running time in Figure 4.

Tangram achieved slightly better performance compared with Spark for word count, but is 2x faster for TF-IDF. For fair comparison, we ensured that Spark does not write intermediate results to disk before shuffle. Both Tangram and Spark have high CPU utilization (over 80% for all cores overtime) and low disk utilization (less than 20% at most) for the two applications, similar to the results reported in [44]. We also tested the systems on a faster 10-Gbps network and on a single machine, and Tangram's performance advantage over Spark on TF-IDF is consistent in both settings, which shows that network communication is not the key factor that affects the performance of the systems on TF-IDF. We believe the language (C++ vs. Scala) and other system overheads are the main reasons for the performance difference. Tangram also achieves almost linear scaling when increasing dataset size.

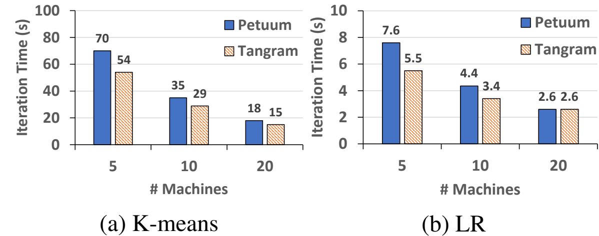
Iterative Machine Learning. For iterative machine learning workloads, we tested K-means [54] and SGD based logistic regression (LR). We used a dense dataset (mnist8m [33]) for

[6] https://en.wikipedia.org/wiki/Tf-idf



(a) Word Count (b) TF-IDF

Figure 4: Running time for word count and TF-IDF



(a) K-means (b) LR

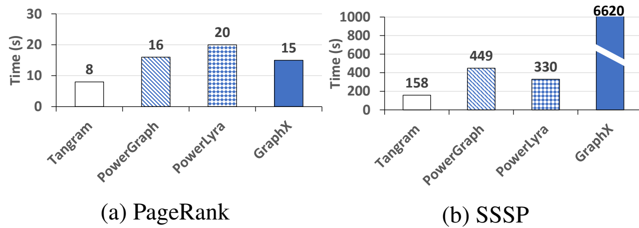
Figure 5: Per-iteration time for K-means and LR

K-means and a sparse dataset (webspam [59] with a sparsity of 2.24×10^{-4}) for LR, in order to test Tangram's performance under different data sparsity. We replicated the datasets 10 times for better scalability tests. BSP was used for K-means, while SSP ($s = 2$) was used for LR. We compared Tangram with the state-of-the-art parameter server system, Petuum Bösen [60]. We did not compare with Spark as it has been shown to be inefficient for iterative machine learning workloads compared with Petuum [29, 62].

Figure 5 reports the per-iteration time obtained by averaging 20 iterations, while varying the numbers of machines. Tangram's performance is very competitive compared with Petuum, as Tangram also supports optimizations generally used in parameter server based systems, such as process cache and message combining (Section 5.5). The scaling performance of Tangram and Petuum is better for K-means than for LR since K-means is CPU-bound, while LR is network-bound due to the large model.

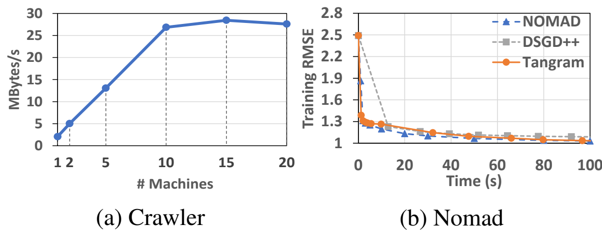
Graph Analytics. For graph analytics workloads, we compared with GraphX [22], PowerGraph [21] and PowerLyra [13]. GraphX is built on Spark and adopts immutable data abstraction, while PowerGraph and PowerLyra use mutable abstraction and support fine-grained state access. We tested PageRank and single source shortest path (SSSP), in BSP mode. We used the webuk graph [7], which has 133M vertices and 5.5B edges.

We report the per-iteration time for PageRank and the total running time for SSSP in Figure 6. Tangram achieves better performance than even the specialized systems. We found that this is because both PageRank and SSSP are network-bound, and message combining in Tangram (edge-cut + delay combiner) is more effective in reducing communication than other systems (vertex/hybrid-cut + combiner). GraphX outperforms PowerGraph and PowerLyra on PageRank as the workload is heavy and balanced in each iteration. In contrast, the access



(a) PageRank (b) SSSP

Figure 6: Comparison on PageRank and SSSP



(a) Crawler (b) Nomad

Figure 7: Performance of distributed crawler and Nomad

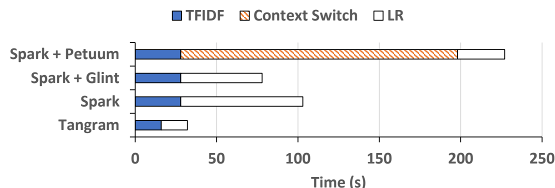


Figure 8: Completion time of the pipelined workload

pattern of SSSP is more sparse, and thus the lack of support for in-place updates renders GraphX inefficient.

Other Workloads. We also evaluated the performance of Tangram on a wider variety of computation patterns using distributed crawler and Nomad [69].

Figure 7a reports the download speed of Tangram-based crawler. The download speed of the crawler scales almost linearly with the number of machines and quickly consumes the download bandwidth of the whole cluster and reaches a plateau, which is similar to Piccolo-based crawler [46].

Nomad is an efficient SGD-based asynchronous algorithm for matrix factorization (MF) and has a complex computation pattern that migrates item latent factors among machines. We used the Yahoo! Music dataset [63] and compared Tangram-based Nomad with MPI-based Nomad [69] and DSGD++ [56] (another state-of-the-art MF algorithm). Figure 7b reports their training root mean square errors (RMSE). Tangram performs slightly worse than MPI-based Nomad initially but catches up later. Compared with MPI-based DSGD++, Tangram has better performance most of the time. Although the MPI-based implementations are efficient, Tangram offers very competitive performance and more user-friendly API.

Pipelined Workload. We implemented a simple pipelined workload that computes TF-IDF vectors with 2^{18} features from the 50GB English Wikipedia dataset and trains an LR model using gradient descent for 30 iterations. We compared

Table 3: Overhead of Task Management & Progress Control

App	Controller CPU %	Bitmap size
WordCount	1.12%	636KB
PageRank	0.99%	638KB
LR	0.29%	341KB
K-means	0.02%	4KB
Nomad	3.38%	153KB

Tangram with Spark, Spark + Glint [28], and Spark + Petuum. Glint is a built-in parameter server for Spark, and thus Spark + Glint uses Spark for TF-IDF and Glint for LR. Spark + Petuum uses Petuum for LR. Other systems (e.g., Naiad [39]) can also handle such a pipeline, but they mainly target at streaming workloads and have more expensive fault tolerance mechanisms. This experiment was conducted using a faster 10-Gbps network but the relative performance of the systems is consistent when running on a 1-Gbps network.

We report the execution time of the pipeline in Figure 8. Tangram used much less time than Spark mainly because Spark is not efficient for machine learning workloads [29, 62]. However, Spark + Petuum took even longer time, even though Petuum was much faster than Spark (29 seconds vs. 75 seconds) for LR. This is because there is a costly context switch overhead when moving from Spark to Petuum (around 170 seconds for dumping and loading the 45GB TF-IDF vectors) [7]. Spark + Glint removes the context-switch overhead and the performance is slightly better than Spark. However, adding dependencies (e.g., Glint) in Spark violates Spark's unified abstraction and breaks Spark's fault tolerance semantics.

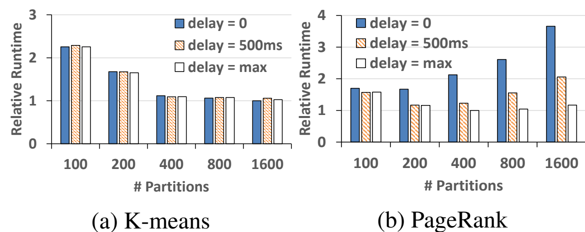
We also tested Flink [10], but LR was an order of magnitude slower on Flink compared with Spark [8]. Thus, we do not report the details of Flink's results. In comparison to these popular systems, the performance of Tangram in Figure 8 demonstrates its benefits as a general and efficient system for processing pipelined workloads.

6.3 Evaluation of System Designs

This set of experiments evaluates the effects of the system designs on the performance of Tangram. We first examine the average CPU consumption of the local controller on each worker and the total size of the bitmap used for progress control. Table 3 shows that the CPU consumption of the local controller is consistently low and the bitmap has a small memory footprint for all workloads. Nomad has the highest controller overhead as the algorithm needs to handle the frequent migration of item latent vectors. K-means has the smallest bitmap size because a single partition is used for the centers. In general, both controller CPU consumption and bitmap size increase with the number of partitions. Empirically, setting the number of partitions to 1-3 times the number

[7] Although in-memory caching systems like RAMCloud [41] or optimized distributed file systems like Tachyon [30] may reduce the context switch cost, the cost of dumping and loading the datasets is still non-negligible.

[8] The per-iteration time of LR using Flink Machine Learning library and using Flink DataSet API is 97x and 21x of that of Spark, respectively.



(a) K-means (b) PageRank

Figure 9: Effects of partitioning and delay combiner

of cores achieves good performance. Thus, the cost of local task control and progress management is acceptable.

Next we examine the effect of the number of partitions. Figure 9 shows that the per-iteration time (as a ratio to the optimal setting) of K-means first decreases and is then stabilized as the number of partitions increases. This is because increasing the number of partitions improves the parallelism, until the cores are fully occupied (the cluster has 480 virtual cores). Beyond that point, using more partitions does not improve performance. For PageRank, the per-iteration time first slightly decreases and then increases with the number of partitions. This is because PageRank is network-bound and thus using more partitions results in more communication. The delay combiner in Section 5.5 can be used to reduce the communication overhead by merging the map outputs from multiple local partitions. By setting the combine timeout to maximum, the per-iteration time of PageRank stays almost constant when increasing the number of partitions beyond 400. For K-means, as it is CPU-bound due to a small number of parameters, using the delay combiner has almost no effect.

7 Related Work

Programming Model. MapReduce [17] has inspired the development of many data-parallel analytics frameworks with two coarse-grained high-order functions, *map* and *reduce*. Map-Reduce-Merge [67] extended MapReduce with a Merge phase to support the join of multiple heterogeneous datasets. HaLoop [9], Twister [18] and Map-Reduce-Update [8] adapted MapReduce for iterative computation. Other frameworks, e.g., Dryad [27], FlumeJava [11], Spark [70], Tez [52], etc., generalized the coarse-grained functional model by introducing dataflow graph, which enables easy construction of pipelines involving multiple stages. CIEL [40] and Ray [38] further support dynamic task graph. Flink [10] and Naiad [39] support the dataflow model on top of their streaming execution engines. Tensorflow [1] adopts dataflow graph to represent machine learning pipelines.

Some systems adopt the distributed shared memory (DSM) model. Piccolo [46] allows user-defined kernels to read and update distributed key-value tables in parallel. Parameter server systems, e.g., DistBelief [16], Project Adam [14], Parameter Server [31], Petuum [60,62], FlexPS [25], incorporate machine learning specialized optimizations such as bounded delay execution. DSM is flexible but the push/pull API is considered more low-level than the functional API in the dataflow model. Husky [66] adopts an object-oriented API to model different computational frameworks. GraphLab [34, 35] uses a data graph to represent computational structure and data dependencies for some machine learning problems. Graph processing frameworks [12, 15, 21, 22, 34, 36, 50, 51, 61, 64, 65, 74] usually expose vertex/edge/subgraph-centric programming models and incorporate graph specific optimizations.

Execution Model. Popular dataflow systems, e.g., Tez [52], DryadLINQ [68], and Spark [70], adopt a BSP execution model, in which a stage waits for its predecessors to finish. Specialized systems often adopt execution models tailored for their target workloads. GraphLab [34, 35] allows asynchronous vertex execution and uses distributed locking to resolve access conflict. Parameter server systems, e.g., Parameter Server [31], Petuum [60], adopt a bounded delay model (SSP) in which the progress differences among workers are bounded by a user-defined threshold. Maiter [73] and PowerGraph [21] support asynchronous execution optimized for graph workloads. In comparison, Tangram supports BSP, SSP and ASP, enabling it to efficiently process various types of workloads such as graph analytics, machine learning, etc.

Scheduling Model. Recent work [37, 43, 45, 53] observed that centralized scheduling is the bottleneck for scaling out when there are a large number of short-lived tasks. To reduce the control plane overhead, Drizzle [57] and Nimbus [37] cache the scheduling decision, while MonoSpark [42] and Canary [48] use local/distributed scheduler. Tangram avoids the centralized scheduling overhead by relying on the local controllers to schedule their own tasks. The global scheduler only launches plans and manages progress.

8 Conclusions

We proposed a programming model called MapUpdate to determine data mutability according to workloads, which not only brings good expressiveness but also enables a rich set of system features (e.g., asynchronous execution) and provides strong fault tolerance. We developed Tangram to support MapUpdate with novel designs such as partition-based progress control and context-aware failure recovery. We also incorporate optimization techniques such as process cache and partition migration. Our experiments show that Tangram is expressive and efficient, and achieves comparable performance with specialized systems for a wide variety of workloads. Our work demonstrates that we do not have to choose either mutable or immutable abstraction, but can embrace both of them in one unified framework to enjoy the best of both worlds.

Acknowledgments. We thank the reviewers and our shepherd Animesh Trivedi for their constructive comments that help improved the quality of the paper. This work was supported in part by ITF 6904945, and GRF 14208318 & 14222816.

References

[1] 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. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, pages 185–198, 2013.

[4] G. Ananthanarayanan, M. C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: trimming stragglers in approximation analytics. In *NSDI*, pages 289–302, 2014.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010.

[6] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *SIGMOD*, pages 601–613, 2018.

[7] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[8] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *SIGPLAN*, pages 363–375, 2010.

[12] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12, 2018.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[14] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. C. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[19] Enwiki Dump. https://dumps.wikimedia.org/enwiki/.

[20] Flink Parameter Server Limitations. https://github.com/gaborhermann/flink-parameter-server#limitations.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[23] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *SoCC*, pages 98–111, 2016.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[25] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng. Flexps: Flexible parallelism control in parameter server architecture. *PVLDB*, 11(5):566–579, 2018.

[26] IndexedRDD for Apache Spark. https://github.com/amplab/spark-indexedrdd.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[28] R. Jagerman, C. Eickhoff, and M. de Rijke. Computing web-scale topic models using an asynchronous parameter server. In *SIGIR*, pages 1337–1340, 2017.

[29] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478, 2017.

[30] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, pages 6:1–6:15, 2014.

[31] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[32] libhdfs3. https://github.com/Pivotal-DataFabric/attic-libhdfs3.

[33] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 301–320. MIT Press, Cambridge, MA., 2007.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[36] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[37] O. Mashayekhi, H. Qu, C. Shah, and P. Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *ATC*, pages 513–526, 2017.

[38] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577, 2018.

[39] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[40] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[41] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[42] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, pages 184–200, 2017.

[43] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *HotOS*, 2013.

[44] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

[45] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[46] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 293–306, 2010.

[47] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *PVLDB*, 11(12):1876–1888, 2018.

[48] H. Qu, O. Mashayekhi, C. Shah, and P. Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *EuroSys*, pages 1:1–1:13, 2018.

[49] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[50] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[51] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[52] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. C. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, pages 1357–1369, 2015.

[53] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, pages 351–364, 2013.

[54] D. Sculley. Web-scale k-means clustering. In *WWW*, pages 1177–1178, 2010.

[55] Standby Masters with ZooKeeper. https://spark.apache.org/docs/latest/spark-standalone.html#standby-masters-with-zookeeper.

[56] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *ICDM*, pages 655–664, 2012.

[57] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, pages 374–389, 2017.

[58] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[59] S. Webb, J. Caverlee, and C. Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically. In *CEAS*, 2006.

[60] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, pages 381–394, 2015.

[61] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[62] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[63] Yahoo! Webscoope. http://webscope.sandbox.yahoo.com/.

[64] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[65] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[66] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[67] H. Yang, A. Dasdan, R. Hsiao, and D. S. P. Jr. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[68] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[69] H. Yun, H. Yu, C. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[72] ZeroMQ. http://zeromq.org/.

[73] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.*, 25(8):2091–2100, 2014.

[74] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model

Jin Kyu Kim[1] Abutalib Aghayev[1] Garth A. Gibson[1,2,3] Eric P. Xing[1,4]
[1]*Carnegie Mellon University* [2]*Vector Institute* [3]*University of Toronto* [4]*Petuum, Inc.*

Abstract

It is a daunting task for a data scientist to convert sequential code for a Machine Learning (ML) model, published by an ML researcher, to a distributed framework that runs on a cluster and operates on massive datasets. The process of fitting the sequential code to an appropriate programming model and data abstractions determined by the framework of choice requires significant engineering and cognitive effort. Furthermore, inherent constraints of frameworks sometimes lead to inefficient implementations, delivering suboptimal performance.

We show that it is possible to achieve *automatic* and *efficient* distributed parallelization of *familiar* sequential ML code by making a few mechanical changes to it while hiding the details of concurrency control, data partitioning, task parallelization, and fault-tolerance. To this end, we design and implement a new distributed ML framework, STRADS-Automatic Parallelization (AP), and demonstrate that it simplifies distributed ML programming significantly, while outperforming a popular data-parallel framework with a *non-familiar* programming model, and achieving performance comparable to an ML-specialized framework.

1 Introduction

The systems community has made significant progress on simplifying distributed parallel programming, producing many high-level frameworks such as MapReduce [13], Spark [54], Pregel [34], PowerGraph [19], GraphX [20], PyTorch [40], and TensorFlow [2]. To automatically parallelize computation while achieving essential requirements such as fault tolerance and load balancing, these frameworks offer constrained programming models and limited data abstractions. For example, Spark offers Resilient Distributed Datasets (RDDs) without fine-grained write access; Spark and MapReduce ask programmers to specify a program using a handful of operators such as *map* and *reduce* while GraphLab requires adopting a rarely used *vertex-centric* programming model.

The programming models of these frameworks are different from a sequential programming model that is widely taught and easily understood [27]. Therefore, it is not surprising that rewriting sequential ML code using the data abstractions and programming models provided by the frameworks incurs significant effort. Furthermore, the simplicity of the mechanisms provided can often result in suboptimal use of cluster resources. These frameworks abstract away data placement, task mapping, and communication, which comes at the cost of limited access to hardware resources, and challenge in implementing ML algorithms efficiently. Studies show that a single threaded [35] or an MPI implementation [50] of popular ML algorithms is up to two orders of magnitude faster than the corresponding implementations on popular frameworks. In summary, a high-level framework often requires data scientists to switch to a **different mental programming model** with its own peculiarities, and it can end up delivering **suboptimal performance**. We believe that the complexity surrounding distributed ML programming as well as the inefficiency in execution are *incidental* and not *inherent*. That is, many sequential ML code can be automatically parallelized to make near optimal use of cluster resources. To prove our point, we present STRADS-AP, a novel distributed ML framework that provides an API requiring minimally-invasive, mechanical changes to sequential ML program code, and a runtime that automatically parallelizes the code on an arbitrary-sized cluster while delivering the performance of hand-tuned distributed ML programs.

STRADS-AP is an evolution of STRADS [26] that provides a framework for parallelizing the execution of ML programs according to user-specified scheduling plan. The plan usually avoids data conflicts, thereby improving statistical progress per iteration. The challenge with STRADS is that the user needs to understand the code and manually come up with a scheduling plan. STRADS-AP addresses this challenge by automatically generating data conflict-free scheduling plan.

STRADS-AP's API frees data scientists from the challenge of molding sequential ML code to a framework's programming model. To achieve this, the STRADS-AP API offers Distributed Data Structures (DDSs), such as *vector* and *map*, that allow fine-grained read/write access to elements, as well as two familiar loop operators. During runtime, these loop operators parallelize loop bodies over a cluster following two popular ML parallelization strategies: asynchronous parallel execution, and synchronous parallel execution, with strong or relaxed consistency.

STRADS-AP's workflow, shown in Figure 1, starts with a data scientist making mechanical changes to sequential code (Figure 1(a, b).) The code is then preprocessed by STRADS-AP's preprocessor and complied into binary code by a C++ compiler (Figure 1(c).) Next, STRADS-AP's runtime executes the binary on nodes of a cluster while hiding details of distributed programming (Figure 1(d).) The runtime system is responsible for (1) transparently partitioning DDSs that store training data and model parameters, (2) parallelizing slices of ML computations across a cluster, (3) fault-tolerance, and (4) enforcing strong consistency on shared data if required, or synchronizing partial outputs with relaxed consistency.

To fill the gap of debugging tools for distributed ML pro-

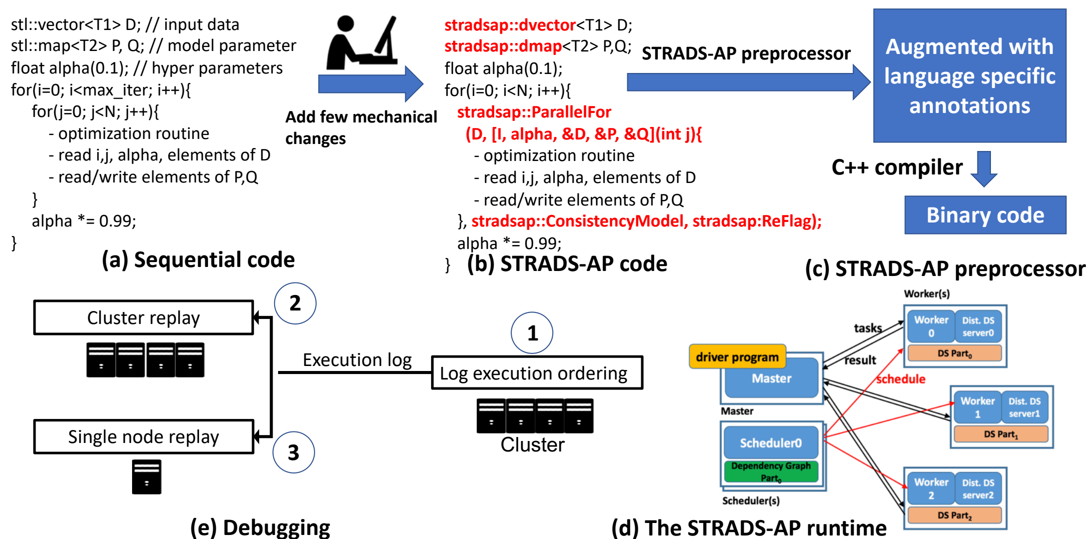


Figure 1: STRADS-AP workflow: (a) Data scientist implements an ML algorithm in sequential code; (b) Derives STRADS-AP parallel code with mechanical changes; (c) STRADS-AP preprocessor adds more annotation to address language-specific constraints, and the source code is compiled by a native compiler; (d) The STRADS-AP runtime runs the binary in parallel on a cluster; (e) Debugging features of STRADS-AP: Logging parallel execution order, and replaying it on a cluster ② for deterministic re-execution, or on a single node ③ for easy debugging.

grams, STRADS-AP offers two debugging modes—*cluster replay* and *single-node replay*—as shown in Figure 1(e). In *cluster replay* mode, the parallel execution log from the previous parallel run is replayed by obeying the same lock grant ordering and message ordering (② in Figure 1(e)), allowing deterministic re-execution. In *single-node replay* mode, the parallel execution log is replayed on a single node (③ in Figure 1(e)) allowing easier inspection of program state with a debugger.

TensorFlow and PyTorch simplify programming Deep Neural Networks (DNN) models, which is just one of the plethora of ML models. Implementing or researching non-DNN models and algorithms in these frameworks, however, often requires adding new kernel operators for parallelization, taking significant effort (Section 6.1). STRADS-AP, on the other hand, provides automatic parallelization of a wide range of non-DNN algorithms by requiring few mechanical changes to a serial implementation.

We implement STRADS-AP as a C++ library in about 16,000 lines of code.[1] STRADS-AP is largely rewritten from scratch, reusing some components of STRADS. We evaluate its performance on a moderate-sized cluster with four widely-used ML applications, using real data sets. To evaluate the increase in user productivity, we ask a group of students to convert a serial ML application to a distributed one using STRADS-AP, and we report our findings. The key contributions of our work are:

- The STRADS-AP API, a familiar C++ STL-like data structures and loop operators, requiring minimal changes when converting sequential ML code to STRADS-AP parallel code.
- The STRADS-AP runtime that achieves low latency DDS access, fault-tolerance, and concurrency control.
- Two debugging modes that simplify debugging and verification of distributed ML programs.
- Performance and productivity evaluation with four well-established ML applications implemented on STRADS-AP.

In the rest of this paper, we first make the case for STRADS-AP by presenting the complications imposed by high-level frameworks on users (Section 2), as well as the performance bottlenecks caused by their simple mechanisms, giving specific examples. We then present the STRADS-AP API (Section 3), and runtime implementation details (Section 4). Next, we give an overview of STRADS-AP's debugging features (Figure 5), followed by an extensive performance and productivity evaluation (Section 6). Finally, we cover the related work (Section 7) and conclude (Section 8).

2 The Cost of Using a Framework

In this section, we demonstrate that converting sequential ML code into high-level framework code requires substantial programming effort and leads to suboptimal performance

Algorithm 1 Pseudocode for SGDMF

1: A: a set of ratings. Each rating contains (i:user id, j:item id, r: rating)
2: W:$M \times K$ matrix; initialize W randomly
3: H:$N \times K$ matrix; initialize H randomly
4: for each rating r in A
5: err = r.r - W[r.i]$*H$[r.j]
6: $\Delta W = \gamma *$(err$*H$[r.j] -$\lambda * W$[r.i])
7: $\Delta H = \gamma *$(err$*W$[r.i] -$\lambda * H$[r.j])
8: W[r.i] += ΔW
9: H[r.j] += ΔH

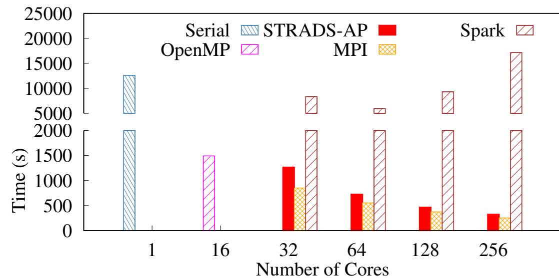
[1] Reported by CLOC [1] tool, skipping blanks and comments.

```
struct rate{int i, int j, float r};
typedef rate T1;
typedef array<float, K> T2;
vector<T1> A = LoadRatings(Datafile_Path);
vector<T2> W(M); RandomInit(W);
vector<T2> H(N); RandomInit(H);
float gamma(.01f), lambda(.1f);
for(int i=0;i<maxiter;i++){
 for(int j=0; j<A.size(); j++){
   const T1 &r = A[j];
   T2 err = r.r - W[r.i]*H[r.j];
   T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j]);
   T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i]);
   W[r.i] += Wd;
   H[r.j] += Hd;
 }
}
```

(a) Sequential SGDMF code

```
struct rate{int i, int j, float r};
typedef rate T1;
typedef array<float, K> T2;
float gamma(.01f), lambda(.1f);
vector<mutex> WLock(M), HLock(N);
for(auto i(0);i<maxiter;i++){
  #pragma omp parallel for
  for(int j=0; j<A.size(); j++){
    const T1 &r = A[j];
    WLock(r.i).lock() // locks to avoid data race
    HLock(r.j).lock()// on shared W,H matrices
    T2 err = r.r - W[r.i]*H[r.j];
    T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j]);
    T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i]);
    W[r.i] += Wd;
    H[r.j] += Hd;
    HLock(r.j).unlock()
    WLock(r.i).unlock()
    // Note that locks are released in reverse
    // ordering of obtaining to avoid deadlock
  }
}
```

(b) Code in (a) parallelized with OpenMP



(e) Time for 60 iterations with Netflix dataset [24], rank = 1000.

```
struct rate{int i, int j, float r};
typedef rate T1;
typedef array<float, K> T2;
dvector<T1> &A = ReadFromFile(Datafile_Path, parser);
dvector<T2> &W = MakeDVector(M, RandomInit);
dvector<T2> &H = MakeDVector(N, RandomInit);
float gamma(.01f), lambda(.1f);
for(int  i=0;i<maxiter;i++){
 AsyncFor(0, A.size()-1, [gamma, lambda, &A,&W,&H](int j){
   const T1 rate &r = A[j];
   T2 err = r.r - W[r.i]*H[r.j];
   T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j]);
   T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i]);
   W[r.i] += Wd;
   H[r.j] += Hd;
 });
}
```

(c) Code in (a) parallellized with STRADS-AP API

```
1  val P = K  // number of executors
2  val ratings = sc.textFile(rfile, P).map(parser)
3  val blks=sc.parallelize(0 until P, P).persist()
4  val W = blks.map(a->Create_WpSubmatrix(a))
5  var H = blks.map(a->Create_HpSubmatrix(a))
6  var AW = ratings.join(W,P)
7  var AWH = AW.join(H,P).mapPartitions(a->ComputeFunc(a,0))
8  float gamma(.01f), lambda(.1f);
9  for(auto i(0);i<maxiter;i++){
10   for(auto sub(0);sub<P;sub++){ // subiteration
11     val idx = i*P + sub;
12     if(idx > 0){
13       AWH = AW(idx).join(H,P).
14         mapPartitions(a->ComputeFunc(a,subepoch))
15     }
16     AW = AWH(idx).mapPartitions(x->separateAW_Func(x))
17     H = AWH.map(x->separateH_and_Shift_Func(x))
18   }
19 }
20 def ComputeFunc(it:Iterator to AWH){
21   val tmp = ArrayBuffer[type of AWH]
22   for(e <- it){
23     val Ap = e.Ap
24     var Wp = e.Wp
25     var Hp = e.Hp
26     for(auto r: Ap){
27       if(r.2 not belong to Hp)
28         continue //skip if not in Hp item indices
29       val err = r.3 - Wp[r.1]*Hp[r.2]
30       val Wd = gamma*(err*Wp[r.1]-lambda*Hp[r.2])
31       val Hd = gamma*(err*Hp[r.2]-lambda*Wp[r.1])
32       Wp[r.i] += Wd
33       Hp[r.j] += Hd
34     } // end of for(auto r ..
35     tmp += Tuple2(e.key, ((Ap, Wp), Hp));
36   } // end of for(e ..
37   val ret = tmp.toArray
38   ret.iterator
39 } // end of def update_Func
```

(d) Algorithm 1 reimplemented with Spark

Figure 2: SGDMF (Algorithm 1) implemented as sequential code (a), and parallelized using OpenMP (b), STRADS-AP (c) and Spark (d). The code snippets show only the core training routine, and do not include data loading and parsing code. STRADS-AP code requires fewer changes to the sequential code compared to OpenMP code, while achieving efficient distributed parallelism in addition to shared-memory parallelism. Spark, on the other hand, requires a complete reimplementation using its abstractions. STRADS-AP outperforms Spark by more than an order of magnitude (e) and continues to scale up to 256 cores, while Spark stops scaling at 64 cores. Hand-tuned MPI code is faster than STRADS-AP by 22% on 256 cores at the cost of a significantly longer programming and debugging effort.

that is orders of magnitude slower than a STRADS-AP implementation, or a hand-tuned implementation.

As a concrete example, we choose Spark as the framework, and Matrix Factorization (MF) as the algorithm—a popular recommender systems algorithm. First, we write sequential code that solves MF using Stochastic Gradient Descent (SGD), denoted as SGDMF in Algorithm 1. Then, we convert the sequential code into three different parallel implementations—shared-memory, STRADS-AP, and Spark—and compare their performance.

2.1 Programming Effort

MF learns user's preferences over all items from an incomplete rating dataset represented as a sparse matrix $A \in \mathbb{R}^{M \times N}$ where M and N are the number of users and items, respectively. MF factorizes the incomplete matrix A into two low-rank matrices, $W \in \mathbb{R}^{M \times K}$ and $H \in \mathbb{R}^{N \times K}$, such that $W \cdot H^T$ approximates A.

Algorithm 1 iterates through the ratings in the matrix A and for each rating $r_{i,j}$, it calculates gradients $\Delta W[i]$, $\Delta H[j]$ and adds the gradients to $W[i]$, $H[j]$, respectively. The computed parameter values for the rating $r_{i,j}$ are immediately visible when computing the next rating, which is an example of asynchronous computation.

2.1.1 Sequential SGDMF code

Sequential implementation of Algorithm 1 is a direct translation of the pseudocode as shown in Figure 2(a).

2.1.2 OpenMP Parallel SGDMF

We parallelize the sequential code in Figure 2(a) using OpenMP [12] to form a single-node baseline. We make two modifications to the sequential code as shown in Figure 2 (b): annotate the loop with parallel-for pragma to let the OpenMP runtime know what to parallelize, and add mutexes to avoid data race on shared matrices W and H. OpenMP parallelizes the inner loop over loop indices using fork-join model where threads run the loop body with different loop indices and join at the completion.

2.1.3 STRADS-AP Distributed SGDMF

Parallelizing the same sequential code using STRADS-AP is done almost mechanically by (1) replacing serial data structures with STRADS-AP's distributed data structures, and (2) replacing the inner loop with STRADS-AP's AsyncFor loop operator, as shown in Figure 2(c).

Unlike OpenMP code, STRADS-AP code has no explicit locking. The runtime is responsible for addressing data conflicts on matrices W and H while executing the loop body in a distributed setting, relieving users from writing error-prone locking code. With little effort, STRADS-AP achieves efficient distributed parallelism, in addition to shared-memory parallelism.

2.1.4 Spark Distributed SGDMF

Unlike with STRADS-AP, parallelizing the sequential code in Figure 2 (a) with Spark requires significant programming effort as detailed below.

Concurrency Control: Spark lacks concurrency control primitives. Since the inner loop of SGDMF leads to data dependencies when parallelized, we need to implement a scheduling plan for correct execution. Reasoning about concurrency control is application-specific and often requires a significant design effort. For SGDMF, we use the Strata scheduling algorithm by Gemulla [18]. The scheduling code shown in Figure 2(d) (lines 9-19) and the training code (lines 20-40) were abridged to fit the page.

Molding SGDMF to Spark API: Even after handling concurrency, implementing SGDMF in Spark requires substantial programming effort for the following reasons.

First, Spark operators, such as *map*, operate on a single RDD object, while the inner loop body in Figure 2(a) accesses multiple objects: the input data A, and the parameter matrices W and H. To parallelize the inner loop with *map* we need to merge A, W, and H into a single RDD, requiring multiple *join* operations involving costly data shuffling.

Second, merging via *join* operator requires changes to data structures. Since *join* operator works only on RDD[Key,Value] type, we have to replace the vectors A, W, H, in Figure 2(a), with RDD[K,V] where V might be also key-value pair type.

Finally, data movement for concurrency control requires extra *join* and *map* operations. At the end of every subiteration, the Strata scheduling algorithm moves H partitions among nodes, which requires two extra operations for every subiteration: (1) a *map* operation that separates H from the merged RDD and modifies the key field of H, (2) a *join* operation that remerges H and AW into AWH for the next subiteration, as shown in Figure 2(d) (lines 9-19.)

In summary, engineering the Spark implementation of SGDMF algorithm involves a large amount of *incidental* complexity that stems from the limitations of Spark API and its data abstractions. As we show next, in addition to the loss in productivity, there is also a loss in efficiency.

2.2 Performance Cost

As a baseline for the distributed implementations, we implement SGDMF using MPI [16] and OpenMP, which are efficient at the cost of larger programming effort. MPI SGDMF uses the same Strata scheduling algorithm and point-to-point communication to circulate H partitions among nodes. All SGDMF implementations achieve proper concurrency control, making similar statistical progress per iteration. Therefore, our performance comparison focuses only on elapsed time for running 60 iterations, after which all implementations converge. We run experiments with Netflix dataset using up to 256 cores on 16 machines that are connected via 40Gbps Ethernet.

As Figure 2(e) shows, Spark is about 68$\times$ slower than MPI on 256 cores. In the same setting, STRADS-AP is slower than

	Requires Changing Programming Model	Programming Language	Application-Level Concurrency Control	Hides Details of Distributed Programming	Fault Tolerance	Debuggability for Distributed ML
STRADS-AP	No	C++	Yes	Yes	Yes (Fast re-execution)	Yes
STRADS [26]	Yes (model-schedule)	C++	Yes(user defined schedule)	Partly (communication)	Yes (Checkpoint)	No
Orion [49]	No	Julia	Yes	Yes	Yes (Checkpoint)	No
GraphLab [19]	Yes (vertex-centric)	C++	Yes	Yes	Yes (Checkpoint)	No
Spark [54]	Yes (map/reduce/...)	Multi	No	Yes	Yes (RDD)	Partly
TensorFlow [2]	Yes (data-flow)	Multi	No	Yes	Yes (Checkpoint)	Yes
Parameter Server [31]	Yes (key-value)	C++	No	Partly (parameter comm)	Yes (Replication)	No
MPI [16]	Yes (message-passing)	C	No	Partly (communication)	No	No
UPC [14]	Yes (PGAS)	C	Partly (lock APIs)	Yes	No	No

Table 1: Summary of features of frameworks used in distributed ML programming. For detailed comparison, refer to Section 2.3 and Section 7. For efficiency comparison, see Section 6

MPI by only 22%, whereas it is over 50× faster than Spark. The suboptimal performance of Spark implementation is due to the aforementioned factors (Section 2.1.4). STRADS-AP is 38.8 and 4.6 times faster than sequential and OpenMP, respectively.

2.3 Other High-Level Frameworks

Our findings of incidental complexity and suboptimal performance are not limited to the example of Spark and SGDMF. For example, PowerGraph provides concurrency control mechanisms, but its *vertex-centric* programming model requires users to redesign data structures to fit to a graph representation and express computations using GAS (Gather, Apply, Scatter) routines. TensorFlow provides a very high-level programming model taking a loss function and automates the gradient update process but does not support serializable asynchronous computation well. Parameter Servers (PS) [3,11,31,48] abstract away the details of parameter communication through the key-value store interface but many other details of distributed parallel programming, such as data partitioning, parallelization of tasks, and application-level concurrency control, are left to the user; that is, PS does not provide an illusion of sequential programming. UPC [14] extends the C programming language with Partitioned Global Address Space (PGAS) programming model that burdens the programmer with the job of doing careful performance tuning (i.e. affinity between threads and shared memory partitions, low-level data layout, use of collective functions such as gather, scatter, reduce.) As Table 1 shows, STRADS-AP and Orion [49] are the only frameworks that allow users to take their sequential code and automatically parallelize it to run on a cluster without sacrificing productivity or efficiency. STRADS-AP owes this flexibility to its familiar API and data structures that we describe next. The differences between STRADS-AP and Orion are described in detail in Section 7.

3 STRADS-AP Programming Interface

STRADS-AP targets ML applications with a common structural pattern consisting of two parts: (1) pretraining part that initializes the model and input data structures, and performs coarse-grained transformations; (2) training part that iteratively optimizes the objective function using nested loop(s) where inner loop(s) perform optimization computations.

To implement a STRADS-AP application, a user writes a simple *driver program* that declares hyper-parameters, invokes operators to create and transform DDSs (Figure 3 (a)), and then invokes STRADS-AP loop operators for optimization (Figure 3(b).) We describe each of these in the following subsections.

```
Create and initialize data structures D for input data
Create and initialize data structures P for model parameters
// run transformations on input data or parameter if necessary
Create and initialize hyper parameters V to control training
```

(a) Pretraining part

```
for(i=0; i<maxiter; i++){// outer loop
  for(j=0; j<N; j++){// inner loop
    // Computations for optimization happen here
    Read a part of input data D
    Read hyper parameters V and loop indices i,j
    Read/writes to a part of model paraemters P
  }
  change hyper parameters
  if(stop condition is true)
    break;
}
```

(b) Training part

Figure 3: ML applications targeted by STRADS-AP are divided into two parts: (a) pretraining part and (b) training part.

3.1 Distributed Data Structures (DDSs)

Table 2 shows a subset of STRADS-AP programming interface. DDS[T] is a mutable in-memory container that partitions a collection of elements of type T over a cluster. DDSs provide a global address space abstraction with fine-grained read/write access and uniform access model independent of whether the accessed element is stored in a local memory or in the memory of a remote node. STRADS-AP offers three types of containers: *dvector*, *dmap*, and *dmultimap*, with interfaces similar to their C++ STL counterparts. These DDSs allow all threads running on all nodes to read and write arbitrary elements while unaware of details such as data partitioning and placement.

Support for distributed and fine-grained read/write accesses gives STRADS-AP an important advantage over other frameworks. It allows reuse of data structures and routines from a sequential program by changing just the declaration of the data type. We describe the inner workings of DDSs in Section 4.3.

3.2 STRADS-AP Operators

The two parts of ML applications, pretraining and training (Figure 3), have different workload characteristics. Pretraining is data-intensive, non-iterative, and embarrassingly-parallel,

	Type	Description
Distributed Data Structures (DDSs)	dvector[T]	A distributed vector of type T elements with per-element read/write access
	dmap[K,V]	A distributed map of [K,V] element pairs of type K and V with per-element read/write access
	dmultimap[K,V]	A distributed multimap of [K,V] element pairs of type K and V with per-element read/write access
Loop Operators	AsyncFor(int64 **S**, int64 **E**, UDF **F**)	Parallelizes closure **F** over indices [**S**, **E**] in isolated manner—avoid ing data conflicts
	SyncFor(DDS[T] **&D**, int **M**, UDF **F**, SyncOpt **S**, bool **RE**)	Parallelizes closure **F** over minibatches of **D** each of size **M** using synchronization option **S** in data-parallel manner. **RE** indicates whether to perform Reconnaissance Execution(§ 4.2)

Table 2: A subset of STRADS-AP API—DDSs, and loop operators for ML training.

whereas training is compute-intensive and iterative, and the inner loop(s) may have data dependencies. STRADS-AP provides two sets of operators that allow natural expression of both types of computation.

3.2.1 Pretraining Operators

STRADS-AP provides *Map*, *Reduce*, *Join*, *Load*, and *Create* operators for loading, storing, and creating DDSs during pretraining. However, STRADS-AP puts no constraints on their usage for expressing training computations.

3.2.2 Loop Operators

STRADS-AP provides loop operators shown in Table 2 to replace the inner loop(s) in the training part of ML programs (Figure 3 (b)). The loop operators take a user-defined closure as the loop body. The closure is a C++ lambda expression that captures the specified DDSs and variables in the scope, and implements the loop body by reading from and writing to arbitrary elements of the captured DDSs. This allows users to mechanically change the loop body of a sequential ML program to STRADS-AP code that is automatically parallelized.

STRADS-AP supports four models of parallelizing ML computations: (1) serializable asynchronous [33], (2) synchronous (BSP [46]), (3) stale-synchronous (SSP [23]), and (4) lock-free asynchronous (Hogwild! [38]) within a node and synchronous across nodes, which we call Hybrid.

STRADS-AP offers two loop operators to support these models. A user can choose AsyncFor loop operator for serializable asynchronous model. For the remaining models a user can choose SyncFor operator and specify the desired model as an argument to the loop operator, as shown in Table 2. Other than choosing the appropriate loop operator, a user does not have to write any code for concurrency-control—the STRADS-AP runtime will enforce the chosen model as described next.

AsyncFor parallelizes the loop over loop indices and ensures isolated execution of the loop bodies even if loop bodies have shared data. In other words, it ensures serializability: the output of the parallel execution matches the ordering of some sequential execution.

AsyncFor takes three arguments: the start index S, the end index $S+N$, and a C++ lambda expression F. It executes $N+1$ lambda instances, $F(S), F(S+1), \ldots, F(S+N)$ concurrently. At runtime, STRADS-AP partitions the index range $S, \ldots, S+N$ into P chunks of size C, and schedules up to P nodes to concurrently execute F with different indices. A node schedules multiple threads to run C lambda instances allowing arbitrary reads and writes to DDSs.

If the lambda expression modifies a DDS, then data conflicts will happen. Although ML algorithms are error-tolerant [23], some algorithms, like Coordinate Descent Lasso [29, 45], LDA [5, 53], and SGDMF [18, 28] converge slowly in the presence of numerical errors due to data conflicts. Following previous work [26], STRADS-AP runtime improves statistical progress by avoiding data conflicts using data conflict-free scheduling for lambda executions. Figure 2(c) shows an example use of AsyncFor implementing SGDMF.

SyncFor parallelizes the loop over the input data. It splits input data into P chunks, where each chunk is processed by P nodes in parallel. Each node processes its data chunk, updating a local replica of model parameters.

SyncFor takes five arguments: the input data D of type DDS[T], the size of a mini-batch M, a C++ lambda expression F, a synchronization option (BSP, SSP, or Hybrid), and a flag indicating whether it should perform Reconnaissance Execution (Section 4.2). The runtime partitions the input data chunk of a node into L mini-batches of size M (typically L is much larger than the number of threads per node), and then schedules multiple threads to process mini-batches concurrently. A thread executes the lambda expression with a local copy of captured variables, and allows reads and writes only to the local copy while running F. At the end of processing a mini-batch, a separate per-node thread synchronizes the local copy of only those DDSs captured by reference across the nodes, and synchronizes local threads according to the sync option. Figure 4 shows an example use of SyncFor that reimplements Google's Word2vec model [22].

By default, SyncFor performs averaging aggregation of model parameters. Users can override this behavior by registering an application-specific aggregation function to a DDS through RegisterAggregationFunc() method.

4 Implementation

This section covers important details of STRADS-AP implementation: the driver program execution (Section 4.1), Reconnaissance Execution (Section 4.2), DDSs (Section 4.3), Concurrency Control (Section 4.4), and STRADS-AP preprocessor (Section 4.5).

4.1 Execution of Driver Program

In the STRADS-AP driver program, the statements are classified into three categories: sequential statements, STRADS-AP

```
typedef vector<word> T1;
typedef vector<array<float, vec_size>> T2;
dvector<T1> &inputD = ReadFromFile<T1>(path, parser);
dvector<T2> &Syn0 = MakeVector<T2>(vocsize, initrow1);
dvector<T2> &Syn1 = MakeVector<T2>(vocsize, initrow1);
float alpha = 0.025;
int W = 5, N = 10;
vector<int> &dtable = InitUnigramtable();
expTable &e = MakeExpTable();
for (int i = 0; i < maxiter; i++){
  SyncFor(inputD, mini-batchsize,
  [W, N, alpha, e, dtable, &Syn0, &Syn1]
  (const vector<T1> &m){
    for (auto &sentence: m){
      //for each window in setence, pick up W words
      //  for each word in the window
      //    run N negative sampling using dist. table
      //    r/w to N rows of Syn0 and Syn1 tables
    }
  }, Hybrid, false);
}
```

Figure 4: Reimplementing Google's Word2vec model using STRADS-AP API.

data processing statements, and STRADS-AP loop statements. The runtime maintains a state machine with one state per category to keep track of the type of code to execute. A driver node starts the driver program in sequential state, and performs sequential execution locally until the first invocation of a STRADS-AP operator. On a STRADS-AP operator invocation, the state machine switches to the corresponding state and the runtime parallelizes the operator over multiple nodes. At the completion of the STRADS-AP operator, the runtime switches back to sequential state and continues running the driver program locally.

The key challenges of the STRADS-AP runtime design are: (1) full automation of concurrency control when parallelizing loop operators, and (2) reducing the latency of accessing DDS elements located on remote nodes. To address these challenges, STRADS-AP implements Reconnaissance Execution.

4.2 Reconnaissance Execution

The runtime system keeps track of the number of invocations of all loop operators in the driver program. On the first invocation of a loop operator, the runtime starts Reconnaissance Execution (RE)—a *virtual execution* of the loop operator. RE is a read-only execution that performs all reads to DDSs, and discovers read/write sets for individual loop bodies. A read/write *access record* of a loop body is a list of tuples, each consisting of a DDS identifier and a list of read/write element indices.

The runtime uses a read/write set for two purposes: (1) performing dependency analysis and generating a data conflict-free scheduling plan for concurrent execution of loop bodies in the AsyncFor operator, and (2) prefetching and caching of DDS elements on remote nodes for low-latency access during the *real execution*.

For the SyncFor operator, when the parameter access is sparse (that is, a small portion of parameters are accessed when processing a mini-batch), the runtime reduces the amount of data transferred by referring to *access records* of RE. However, in applications with dense parameter access, (that is, most parameters are accessed when processing a mini-batch), RE does not help to improve the performance. Therefore, the SyncFor operator's boolean RE parameter (Table 2) allows users to skip RE and prefetch/cache all elements of DDSs captured by the corresponding lambda expression.

To reduce RE overhead, STRADS-AP runs it once per parallel loop operator in the driver program, and reuses read/write set for subsequent iterations. This optimization is based on two assumptions about ML workloads: (1) *iterativeness*—a loop operator is repeated many times until convergence, and (2) *static control flow*—read/write sets of loop bodies do not change over different iterations. That is, the control flow of the inner loop does not depend on model parameter values. Both assumptions are routinely accepted in ML algorithms [3–5, 8, 17, 24, 28, 30, 41, 45, 51–53, 55].

4.3 Distributed Data Structures

On the surface, a DDS is a C++ class template that provides index- or key-based uniform access operator. Under the hood, the elements of a DDS are stored in a distributed in-memory key-value store as key-value pairs. The key is uniquely composed of the table id plus the element index for *dvector*, and the table id plus the element key for *dmap/dmultimap*. Each node in a cluster runs a server of the distributed key-value store containing the elements of a DDS partitioned by the key hash. The implementation of DDS class template reduces the element access latency by prefetching and caching remote elements based on the access records generated by RE (Section 4.2).

The DDSs achieve fault-tolerance through checkpointing. At the completion of a STRADS-AP operator that runs on DDSs, the runtime takes snapshots of any DDSs that were modified or created by the operator. The checkpoint I/O time overhead is negligible because ML programs are compute-intensive, and the input data DDSs are not checkpointed (except once at creation), as they are read-only.

The traditional approach to checkpointing is to dump the whole program state onto storage during the checkpoint, and load the state from the last successful checkpoint during the recovery. Since an ML program may have an arbitrary number of non-DDS variables (like hyper-parameters), the traditional approach would require users to write boilerplate code for saving and restoring the state of these variables, reducing productivity and increasing opportunities for introducing bugs. Therefore, STRADS-AP takes a different approach to checkpointing that obviates the need for such boilerplate code.

Upon a node failure, STRADS-AP restarts the application program in *fast re-execution* mode. In this mode, when the runtime encounters a parallel operator *op* executing iteration i, it first checks to see whether a checkpoint for op_i exists. If yes, the runtime skips the execution of op_i and loads the DDS state from the checkpoint. Otherwise, it continues *normal execution*. Hence, the state of non-DDS variables are quickly and correctly restored without forcing the users to write extra code.

4.4 Concurrency Control

STRADS-AP implements two concurrency control engines: (1) *serializable engine* for the AsyncFor operator, and (2) *data-parallel engine* for the SyncFor operator. Both engines use the read/write set from Reconnaissance Execution (Section 4.2) for prefetching remote DDS elements, while the serializable engine also uses it for making data conflict-free execution plans.

4.4.1 Serializable Engine for AsyncFor

In the serializable engine, a task is defined as the loop body with a unique loop index value i, which ranges from S to E, where S and E are AsyncFor arguments (Table 2). The serializable engine implements a scheduler module that takes the read/write set from RE, analyzes data dependencies, generates a dependency graph, and generates a parallel execution plan that avoids data conflicts. To increase parallelism, the serializable engine may change the execution order of tasks assuming that any serial reordering of the loop body executions is acceptable. This assumption is also routinely accepted in ML computations [26, 33, 39].

The scheduler divides the loop bodies into N *task groups*, where N is much larger than the number of nodes in a cluster, using an algorithm that combines the ideas of static scheduling from STRADS [26] and connected component-based scheduling from Cyclades [39]. The algorithm allows dependencies within a task group but ensures no dependency across task groups. At runtime, the scheduler places task groups on nodes, where each node keeps a pool of task groups.

To balance the load, serializable engine runs a greedy algorithm that sorts task groups in the descending order of size, and assigns task groups to the node whose load is the smallest so far. Once task group placements are finalized, the runtime system starts the execution of the loop operator.

The execution begins by each node initializing DDSs to prefetch necessary elements from the key-value store into a per-node DDS cache. Then each node creates a user-specified number of threads, and dispatches task groups from the task pool to the threads. All threads on a node access the per-node DDS cache without locking, since each thread executes a task group sequentially, and the scheduling algorithm guarantees that there will be no data conflicts across the task groups. In the case of an excessively large task group, we split it among the threads and use locking to avoid data races, which leads to non-deterministic execution.

To reduce scheduling overhead, the serializable engine caches the scheduling plan and reuses it over multiple iterations based on the aforementioned assumptions (Section 4.2). Hence, the overhead of RE and computing a scheduling plan is amortized over multiple iterations.

4.4.2 Data-Parallel Engine for SyncFor

In the data-parallel engine, a task is defined as the loop body with a mini-batch of D with size M, where D and M are SyncFor arguments (Table 2). Hence, a single SyncFor call generates multiple tasks with different mini-batches. The engine places the tasks on nodes that hold the associated mini-batches, where nodes form a pool of assigned tasks.

Similar to the serializable engine, an execution begins by each node initializing DDSs to prefetch necessary elements from the key-value store into the per-node DDS cache, based on the read/write set from RE, and continues by creating a user-specified number of threads.

Unlike the serializable engine, the threads contain a per-thread cache, and are not allowed to access the per-node cache, since in this case there is no guarantee of data conflict-free access. When a node dispatches a task from the task pool to a thread, it copies the parameter values from the per-node cache to the per-thread cache.

Upon task completion, a thread returns the delta between the computed parameter values and the starting parameter values. The node dispatches a new task to the thread, accumulates deltas from all threads, and synchronizes per-node cache with the key-value store by sending the aggregate delta and pulling fresh parameter values.

The SyncFor operator allows users to choose among BSP, SSP, and Hybrid (Section 3.2.2) models of parallelizing ML computations. The BSP [46] and SSP [23] models are well-known, and our implementation follows previous work. Hybrid, on the other hand, is a lesser-known model [25]. It allows lock-free asynchronous update of parameters among threads within a node (Hogwild! [38]), but synchronizes across machines at fixed intervals. In the Hybrid model, a node creates a single DDS cache that is accessed by all threads without taking locks. When all of the threads complete a single task, which denotes a subiteration, the node synchronizes the DDS cache with the key-value store.

4.5 STRADS-AP Preprocessor

While there exist mature serialization libraries for C++, none of them support serializing lambda function objects. The lack of reflection capability in C++, and the fact that lambda functions are implemented as anonymous function objects [36], make serializing lambda challenging. We overcome this challenge by implementing a preprocessor that analyzes the source code using Clang AST Matcher library [10], identifies the types of STRADS-AP operator arguments, and generates RPC stub code and a uniquely-named function object for each lambda expression that is passed to STRADS-AP operators.

The preprocessor also analyzes the source code to see if it declares DDSs of user-defined types. While DDSs of built-in types are automatically serialized using Boost Serialization library [6], for user-defined types the library requires adding boilerplate code, which is automatically added by the preprocessor.

5 Debugging STRADS-AP Applications

STRADS-AP supports two debugging modes: (1) *cluster replay* mode, and (2) *single-node replay* mode. Currently, STRADS-AP debugging modes support only serializable parallel execution generated by AsyncFor operator whose

Dataset	Workload	Feature	Size	Application	Purpose
Netflix [24]	100M ratings	489K users, 17K movies, rank=1000	2.2 GB	SGDMF	Recommendations
1Billion [9]	1 billion words	Vocabulary size 308K, vector size=100	4.5 GB	Word2Vec	Word Embeddings
ImageNet [43]	285K images	1K classes, 21K features, preprocessed by LCC feature extraction [47]	21 GB	MLR	Multi-Class Classification
FreeBase-15K [7]	483K facts	14,951 entities, 1,345 relations, vector size=100	36 MB	TransE	Graph Embeddings

Table 3: Datasets used in benchmarks.

Application	Serial	OpenMP	MPI	STRADS-AP	TF	Spark
SGDMF	✓	✓	✓	✓		✓
MLR	✓	✓	✓	✓	✓	
Word2vec	✓	✓	✓	✓	✓	
TransE	✓			✓		

Table 4: ML programs used for benchmarking. Serial and OpenMP are single core and multi-core applications on a shared-memory machine, respectively, while the rest are distributed parallel applications. MPI applications use OpenMP for shared-memory parallelism within the nodes.

execution can be non-deterministic (Section 4.4.1). The support for SyncFor operator is in progress.

Cluster Replay: The AsyncFor operator allows non-deterministic execution for achieving high performance. Instead of predefined deterministic execution [32], STRADS-AP logs the execution order including lock grant ordering and message ordering, and allows users to replay the log. For this purpose, STRADS-AP implements record/replay modules. The record module records the ordering of lock grantings in every node, and the ordering of message arrivals in the DDS key-value store into persistent storage. To avoid coordination overhead and bottlenecking a single node, each node records partial ordering locally, without making a total ordering. When replaying the log, each node enforces the same partial order.

Single-node Replay: Debugging an ML program can be classified into two categories: (1) search for a traditional software bug, and (2) the inspection of the optimization path. Unfortunately, these debugging tasks are not easy to do in a distributed environment since step-by-step tracing of a distributed application is hard. To address this problem, STRADS-AP offers *single-node replay* mode for parallel ML applications. The *single-node replay* mode takes a parallel execution log, and replays the ordering in a single node setting, where users can trace the program execution using a debugger like GDB.

6 Evaluation

We evaluate STRADS-AP on (1) application performance, and (2) programmer productivity, using the following real world ML applications: SGDMF, Multinomial Logistic Regression (MLR), Word2vec, and TransE [7], summarized in Table 4.

For performance evaluation, as a baseline we implement these applications as (1) a sequential C++ application, (2) a single-node shared-memory parallel C++ application using OpenMP, and (3) a distributed- and shared-memory parallel C++ application using MPI and OpenMP. We then compare the iteration throughput (time per epoch) and the statistical accuracy of Spark, TensorFlow (TF), and STRADS-AP implementations of these applications to those of the baselines, while running them on real datasets shown in Table 3. For brevity, in the rest of the paper, when we mention that an application is implemented using MPI, we mean that it uses OpenMP on a single node and MPI among the nodes.

For productivity evaluation, we conduct two user studies on a group of students with Word2vec and TransE applications. As a measure of productivity, we count the lines of code produced, and measure the time it took students to convert a serial implementation of the algorithm into a STRADS-AP implementation.

All experiments were run on a cluster with 16 machines each with 64 GB of memory and 16-core Intel Xeon E5520 CPUs, running Ubuntu 16.04 Linux distribution, connected with 40 Gbps Ethernet network. The reported numbers are the averages of at least three runs. Error bars are not included due to low variance among the runs.

6.1 Word2Vec

Word2vec is a Natural Language Processing (NLP) model developed by Google that computes vector representations of words, called "word embeddings". These representations can be used for various NLP tasks, such as discovering semantic similarity. Word2vec can be implemented using two architectures: continuous bag-of-words (CBOW) or continuous skip-gram, the latter of which produces more accurate results for large datasets [22].

We implement the skip-gram architecture in STRADS-AP based on Google's open source multithreaded implementation [22] written in C. We make two changes to Google's implementation: (1) modify it to keep all the input data in memory to avoid I/O during training, and (2) replace POSIX threads with OpenMP. After our changes, we observe 6% increase in performance on 16 cores. We then run our improved implementation using a single thread for the serial baseline, and using 16 threads on 16 cores for the shared-memory parallel baseline.

Google recently released a highly-optimized multithreaded Word2vec [21] implementation on TensorFlow with two custom kernel operators written in C++. As of now, Google has not yet released a distributed version of Word2vec on TensorFlow. Therefore, we extend Google's implementation to run in a data-parallel distributed setting. To this end, we modify the kernel operators to work on partitions of input data, and synchronize parameters among nodes using MPI.

Performance Evaluation: Figure 5 shows the execution time

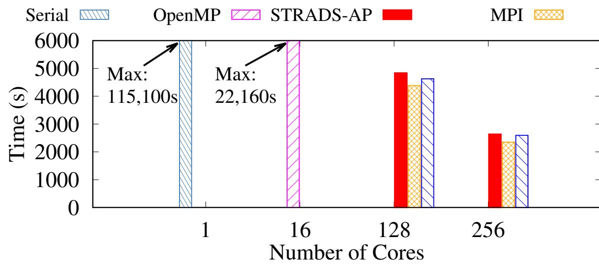


Figure 5: Time for 10 iterations of Word2Vec on 1 Billion word dataset [9] with vector size = 100, window = 5, negative sample count = 10.

Cores	Similarity			Analogy		
	STRADS-AP	MPI	TF	STRADS-AP	MPI	TF
128	0.601	0.601	0.602	0.566	0.564	0.568
256	0.603	0.597	0.601	0.562	0.557	0.561

	Similarity	Analogy
Serial	0.610	0.570
OpenMP	0.608	0.571

Table 5: The top table reports similarity test accuracy [15], and analogy test accuracy [37] for distributed Word2Vec implementations on 1 Billion word dataset, after 10 iterations. The bottom table shows respective values for the serial and OpenMP implementations.

of Word2vec for 10 iterations with a 1 billion word data set. On 256 cores (16 machines), MPI performs better than TensorFlow and STRADS-AP by 9.4% and 10.1%, respectively; that is, a STRADS-AP program obtained by mechanical changes to serial code matches the performance of a highly optimized TensorFlow program. The higher performance of an MPI program stems from the serialization overhead in TensorFlow and STRADS-AP. The MPI implementation stores parameters in arrays of built-in types, and uses in-place MPI_Allreduce call to operate on values directly. STRADS-AP outperforms serial and OpenMP implementations by 45× and 8.7×, respectively.

Table 5 shows the similarity test accuracy [15] and the analogy test [37] accuracy, after running 10 iterations. Using the accuracy of the serial algorithm as the baseline, we see that parallel implementations report slightly lower accuracy (within 1.1%) than the baseline due to the use of stale parameter values.

Productivity Evaluation: Table 6 shows the line counts of Word2vec implementations in the first column. The STRADS-AP implementation has 15% fewer lines than the serial implementation, which stems mainly from the coding style and the use of STRADS-AP's built-in text-parsing library. If we focus the comparison on the core of the program—the training routine—both implementations have around 100 lines, since STRADS-AP implementation takes the serial code and makes a few simple changes to it.

The TensorFlow implementation, however, has three times more lines in the training routine. The increase is due to (1) splitting the training into two kernel operators to fit the

Implementation	Word2vec	MLR	SGDMF
Serial	468	235	271
STRADS-AP	404	245	279
MPI	559	313	409
TensorFlow	646*	155 (Python)	N/A
Spark	N/A	N/A	249 (Scala)

Table 6: Line counts of model implementations using different frameworks. Unless specified next to the lines counts, the language is C++. *TensorFlow implementation of Word2vec has 282 lines in Python and 364 lines in C++.

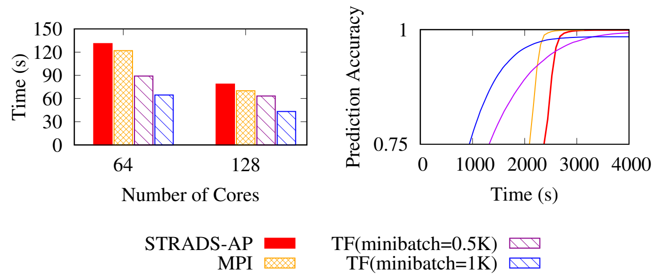


Figure 6: The left figure shows time for a single iteration. We run the TensorFlow implementation with a minibatch sizes of 500 and 1,000. STRADS-AP and MPI implementations do not use vector instructions, therefore, we run them with a minibatch size of 1. Serial and OpenMP implementations (omitted from the graph) also run with a minibatch size of 1, and take 3,380 and 467 seconds to complete, respectively. The right figure shows prediction accuracy as the training progresses. While each implementation runs for 60 iterations, the graph shows only the time until all of them converge to a stable value.

dataflow model, (2) converting tensors into C++ Eigen library matrices and back, and (3) lock management.

While TensorFlow enables users to write simple models easily, it requires a lot more effort and knowledge, which most data scientists lack, to produce high-performance distributed model implementations with custom kernel operators. On the other hand, STRADS-AP allows ordinary users to easily obtain performance on par with the code that was optimized by Google, by making trivial changes to a serial implementation.

6.2 Multinomial Logistic Regression

We implement a serial, OpenMP, MPI, TensorFlow, and STRADS-AP versions of MLR. Our distributed TensorFlow implementation uses TensorFlow parameter servers, and is based on the MNIST code in the TensorFlow repository. Similar to other implementations in our benchmark, our TensorFlow implementation preloads the dataset into memory before starting the training, and uses the Gradient Descent optimizer.

Performance Evaluation: Figure 6 shows single iteration time on 25% of the ImageNet dataset [43] on the left, and accuracy after 60 iterations on the right. TensorFlow makes heavy use of vector instructions, which explains the 30% decrease in runtime when increasing the minibatch size from 500 to 1,000, and almost two times shorter runtime than MPI and STRADS-

Subject	Major (Main PL)	C++ Skill Level	[T1]	[T2]	[T3]	[T4]	[T5]	Total	Challenges
Student 1	Data Mining (Python)	Low	0.25	0.1	0.25	0.1	1.7	2.4	Lack of C/C++ experience
Student 2	Data Mining (Java)	Low	0.3	0.2	0.1	0.2	1.4	2.2	Lack of C/C++ experience
Student 3	Machine Learning (Python)	Low	0.3	0.5	0.5	0.5	1	2.8	Lack of C/C++ experience
Student 4	Compilers (Java)	High	0.3	0.3	0.2	0.1	1.0	1.9	Lack of ML programming familiarity
Student 5	Systems (C++)	High	0.25	0.25	0.5	0.25	0.25	1.5	N/A

Table 7: The breakdown of times (in hours) of five students that converted the serial implementation of the TransE [7] graph embedding algorithm to a distributed STRADS-AP implementation. We split the conversion task into five subtasks: [T1] understand the algorithm, [T2] understand the reference serial code, [T3] review STRADS-AP API guide, [T4] review the provided serial MLR code and the corresponding STRADS-AP code, [T5] convert the serial implementation to STRADS-AP implementation.

AP implementations, which do not use vector instructions. On the other hand, as the right graph in Figure 6 shows, TensorFlow sacrifices accuracy for higher throughput. Unlike the MPI and STRADS-AP implementations that achieve 99.5% accuracy after about 2,800 seconds, the accuracy of TensorFlow remains under 98.4 even after 4,000 seconds. The difference in accuracy is due to STRADS-AP and MPI implementations running with a minibatch size of 1, given that they do not use vector instructions. A single iteration of TensorFlow with a minibatch size of 1 (for which it was not optimized) took about 6 hours, which we omitted from the graph.

Productivity Evaluation: Table 6 shows the line counts of MLR implementations in the second column. The TensorFlow implementation has 38% and 50% fewer lines than the STRADS-AP and MPI implementations, respectively, because while both of the latter implement large chunks of code to compute gradients and apply them to parameters, TensorFlow hides all of these under library function calls. On the other hand, most of the TensorFlow implementation consists of code for partitioning data and setting up the cluster and parameter server variables. This is counter-productive for users who do not want to deal with cluster setup and data partitioning, but want to change the algorithms.

6.3 Matrix Factorization

We already covered (Section 2.1) the implementation details and performance evaluation of solving Matrix Factorization algorithm using SGD optimization (SGDMF). Therefore, we continue with the productivity evaluation.

Productivity evaluation: Table 6 shows line counts of SGDMF implementations in the third column. SGDMF implementation in Scala, even after including the line count for Gemulla's Strata scheduling algorithm (Section 2.1.4), has about 15% fewer lines than STRADS-AP implementation. This is not surprising, given that functional languages like Scala tend to have more expressive power than imperative languages like C++. However, the difficulty of implementing the Strata scheduling algorithm is not captured well in the line count. Figuring out how to implement this algorithm using the limited Spark primitives, and tuning the performance so that the lineage graph would not consume all the memory on the cluster took us about a week, whereas deriving STRADS-AP implementation from the pseudocode took us about an hour.

The line count of MPI is higher than serial code due to Strata scheduling implementation and manual data partitioning.

6.4 User Study

To further evaluate the productivity gains of using STRADS-AP, we conducted two more user studies. In the first study, as a capstone project we assigned a graduate student to implement a distributed version of Word2vec using STRADS-AP and MPI, after studying Google's C implementation [22]. The student had C and C++ programming experience, and had just finished an introductory ML course. After studying the reference source code, the student spent about an hour studying the STRADS-AP API and experimenting with it. It then took him about two hours to deliver a working distributed Word2vec implemented with STRADS-AP API. On the other hand, it took the student two days to deliver a distributed Word2vec implemented with MPI. The MPI implementation was not able to match the STRADS-AP implementation in terms of accuracy and performance until the student had invested two weeks of performance optimizations.

In the second study, we conducted an experiment similar to a programming exam, with five graduate students. We provided the students with a two-page STRADS-AP API documentation, example serial MLR code, and the corresponding STRADS-AP code. We then gave the students a serial C++ program written by an external NLP research group that implemented the TransE [7] graph embedding algorithm, and asked them to produce a distributed version of the same program using STRADS-AP.

Table 7 shows the breakdown of times each student spent at different phases of the experiment, including the students' backgrounds, and the primary challenges they faced. While most students lacked proficiency in C++, they still managed to complete the conversion in a reasonable amount of time. Student 5, who was the most proficient in C++, finished the experiment in 1.5 hours, while Student 1 took 2.4 hours, most of which he spent in the last subtask debugging syntax errors, after breezing through the previous subtasks. Feedback from the participants indicated that (1) converting serial code into STRADS-AP code was straightforward because data structures, the control flow, and optimization functions in the serial program were reusable with a few changes, and (2) the lack of C++ familiarity was the main challenge. The list of reported

mistakes included C++ syntax errors, forgetting to resize local C++ STL vectors before populating them, and an attempt to create a nested DDS, which STRADS-AP does not currently support. We evaluated the students' implementations by running them on FreeBase-15K [7] dataset for 1000 iterations with vector size of 50. The students' implementations were about 22× faster than the serial implementation on 128 cores, averaging at 45.3% accuracy, compared to 46.1% accuracy achieved by the serial implementation.

6.5 Scope and Limitations of STRADS-AP

STRADS-AP facilitates converting serial ML programs into distributed ML programs with minimal changes. Our evaluation shows that the converted ML programs achieve performance comparable to hand-tuned distributed implementations, and to implementations written using ML-specialized frameworks. To achieve higher performance, STRADS-AP relies on two optimizations: reordering of loop indices to find more opportunities for parallelism, and reuse of RE output to amortize the overhead of running RE and making scheduling decision over multiple iterations. These optimizations require ML programs to meet three assumptions: serializability (4.4.1), iterativeness (4.2), and static control flow (4.2). Fortunately, these three assumptions are commonly found in a broad range of ML applications. However, STRADS-AP has some limitations on its expressiveness. For example, it does not support nested parallel loops and user defined data structures having nested DDSs.

7 Related Work

STRADS-AP's design elements rely on a body of previous work. The virtual iteration of IterStore parameter server [11] inspired Reconnaissance Execution (RE). IterStore uses the read/write set only for prefetching parameters into nodes from the parameter server. STRADS-AP, however, uses the read/write set for generating a data conflict-free execution schedule as well as prefetching.

Calvin [44] introduces reconnaissance queries for efficient distributed transactions with low locking overhead. However, Calvin cannot reuse the results of the query because DBMS workloads do not generally have the *iterativeness* and *static control flow* properties of ML workloads. STRADS-AP runtime runs RE on just the first invocation, and the output of RE is reused for every iteration until convergence, amortizing the cost of RE.

OpenMP [12] and Distributed R [42] are popular among ML programmers and provide parallel loop operators. However they lack the support of high-level abstractions to parallelize ML programs in which the ML training routine has data dependencies. For example, when loop bodies of a parallel loop have data dependencies on shared ML model parameters, OpenMP and Distributed R delegate concurrency control to the ML programmer. This requires ML programmers to write routines for aggregating shared parameters in the case of synchronous parallel execution, and handling data dependencies in the case of asynchronous execution. On the contrary, the STRADS-AP runtime hides parameter aggregation and concurrency control from ML programmers through Sync/Async loop operators.

GraphLab [19, 33], Cyclades [39], and STRADS [26, 29] present ML scheduling ideas that avoid executing updates with data conflicts to improve statistical progress per iteration. However, GraphLab expects users to express a serial ML algorithm using GAS (Gather, Apply, Scatter) primitives, Cyclades targets a single shared-memory machine, limiting scalability, and STRADS requires users to design and implement data conflict-free scheduling strategy. STRADS-AP addresses all of these limitations by (1) allowing users to convert a serial program into parallel program through mechanical changes, (2) scaling out to an arbitrary-sized cluster, and (3) automatically generating data conflict-free execution schedules.

More recently, Orion [49] proposed automatic parallelization using static analysis of matrix index access patterns. STRADS-AP and Orion [49] share the same goal of automating scheduling decision. However, while Orion targets ML programs written in Julia scripting language, STRADS-AP targets C++ ML programs because there are a large number of serial ML programs in written in C++. This difference in the choice of programming language leads us to explore substantially different design options, such as STL-like DDSs and dynamic analysis, instead of distributed matrix and static analysis. Specifically, Orion's static analysis requires that data dependencies are determined statically in the form of a linear combination of loop variables. This assumption does not hold frequently in real-world ML applications. On the contrary, STRADS-AP performs dynamic analysis that captures data accesses and dependencies at runtime without relying on such assumptions.

8 Conclusion

Despite the availability of a plethora of frameworks for distributed Machine Learning (ML) programming, we believe distributed ML programming is still unnecessarily complicated. Each framework comes with its own restricted programming model and abstractions, its inefficiencies, and peculiarities that add to the growing list of things that data scientists should master. We take a step back and ask: how can we take a serial imperative implementation of an ML model, and parallelize it over a cluster with minimal effort from the user.

Our answer is STRADS-AP—a distributed ML framework, which is a combination of a runtime and an API comprised of Distributed Data Structures (DDSs) and parallel loop operators. Using four real-world applications, we show that STRADS-AP allows data scientists to easily convert a serial implementation of an ML model to a distributed implementation that achieves performance comparable to hand-tuned MPI and TensorFlow implementations, while outperforming a Spark implementation by more than an order of magnitude.

Acknowledgements

We thank our shepherd Steven Hand and the anonymous reviewers. This research is supported in part by National Science Foundation under awards CCF-1629559, IIS-1563887, and IIS-1617583. We thank the member companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, and Western Digital) for their interest, insights, feedback, and support.

References

[1] CLOC: Count Lines of Code. http://cloc.sourceforge.net/.

[2] Martin 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, 2016.

[3] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

[4] Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee Whye Teh. On Smoothing and Inference for Topic Models. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 27–34, Arlington, Virginia, United States, 2009. AUAI Press.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003.

[6] Boost. Boost C++ Library - Serialization. http://www.boost.org/doc/libs/1_66_0/libs/serialization/doc/index.html.

[7] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2787–2795, USA, 2013. Curran Associates Inc.

[8] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 321–328, USA, 2011. Omnipress.

[9] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. Technical report, Google, 2013.

[10] Clang. AST Matcher Reference. http://clang.llvm.org/docs/LibASTMatchersReference.html.

[11] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Iterative-ness for Parallel ML Computations. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[14] Tarek El-Ghazawi and Lauren Smith. Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[15] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing Search in Context: The Concept Revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414. ACM, 2001.

[16] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[17] J. Friedman, T. Hastie, H. Hofling, and R. Tibshirani. Pathwise Coordinate Optimization. *Annals of Applied Statistics*, 1(2):302–332, 2007.

[18] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[21] Google. TensorFLow Optimized Word2vec. https://github.com/tensorflow/models/blob/master/tutorials/embedding/word2vec_optimized.py.

[22] Google. word2vec. https://code.google.com/archive/p/word2vec/.

[23] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pages 1223–1231, USA, 2013. Curran Associates Inc.

[24] James Bennett and Stan Lanning and Netflix Netflix. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[25] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. Parallelizing Word2Vec in Multi-Core and Many-Core Architectures. *CoRR*, abs/1611.06172, 2016.

[26] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM.

[27] Keith Kirkpatrick. Parallel Computational Thinking. *Commun. ACM*, 60(12):17–19, November 2017.

[28] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[29] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A. Gibson, and Eric P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2834–2842, Cambridge, MA, USA, 2014. MIT Press.

[30] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the Sampling Complexity of Topic Models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 891–900, New York, NY, USA, 2014. ACM.

[31] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, 2014. USENIX Association.

[32] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[33] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[34] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[35] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[36] Microsoft Developer Network. Lambda Expressions in C++. https://msdn.microsoft.com/en-us/library/dd293608.aspx, 2015.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[38] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings*

of the 24th International Conference on Neural Information Processing Systems, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc.

[39] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I. Jordan, Kannan Ramchandran, Chris Re, and Benjamin Recht. Cyclades: Conflict-free asynchronous machine learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2576–2584, USA, 2016. Curran Associates Inc.

[40] 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. In *NIPS 2017 Autodiff Workshopt: The Future of Gradient-based Machine Learning Software and Techniques*, Long Beach, CA, US, December 9, 2017.

[41] István Pilászy, Dávid Zibriczky, and Domonkos Tikk. Fast ALS-based Matrix Factorization for Explicit and Implicit Feedback Datasets. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 71–78, New York, NY, USA, 2010. ACM.

[42] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[43] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[44] 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 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[45] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[46] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[47] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas S. Huang, and Yihong Gong. Locality-constrained linear coding for image classification. In *CVPR*, pages 3360–3367. IEEE Computer Society, 2010.

[48] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM.

[49] Jinliang Wei, Garth A. Gibson, Phillip B. Gibbons, and Eric P. Xing. Automating dependence-aware parallelization of machine learning training on distributed shared memory. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 42:1–42:17, New York, NY, USA, 2019. ACM.

[50] Jinliang Wei, Jin Kyu Kim, and Garth A. Gibson. Benchmarking Apache Spark with Machine Learning Applications. Technical report, Carnegie Mellon University, 2016.

[51] T.T. Wu and K. Lange. Coordinate Descent Algorithms for Lasso Penalized Regression. *The Annals of Applied Statistics*, 2(1):224–244, 2008.

[52] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 765–774, Washington, DC, USA, 2012. IEEE Computer Society.

[53] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1351–1361, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[54] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[55] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workload

Ashraf Mahgoub
Purdue University

Paul Wood
Johns Hopkins University

Alexander Medoff
Purdue University

Subrata Mitra
Adobe Research

Folker Meyer
Argonne National Lab

Somali Chaterji
Purdue University

Saurabh Bagchi
Purdue University

Abstract

Reconfiguring NoSQL databases under changing workload patterns is crucial for maximizing database throughput. This is challenging because of the large configuration parameter search space with complex interdependencies among the parameters. While state-of-the-art systems can automatically identify close-to-optimal configurations for static workloads, they suffer for dynamic workloads as they overlook three fundamental challenges: (1) Estimating performance degradation during the reconfiguration process (such as due to database restart). (2) Predicting how transient the new workload pattern will be. (3) Respecting the application's availability requirements during reconfiguration. Our solution, SOPHIA, addresses all these shortcomings using an optimization technique that combines workload prediction with a cost-benefit analyzer. SOPHIA computes the relative cost and benefit of each reconfiguration step, and determines an optimal reconfiguration for a future time window. This plan specifies when to change configurations and to what, to achieve the best performance without degrading data availability. We demonstrate its effectiveness for three different workloads: a multi-tenant, global-scale metagenomics repository (*MG-RAST*), a bus-tracking application (*Tiramisu*), and an HPC data-analytics system, all with varying levels of workload complexity and demonstrating dynamic workload changes. We compare SOPHIA's performance in throughput and tail-latency over various baselines for two popular NoSQL databases, Cassandra and Redis.

1 Introduction

Automatically tuning database management systems (DBMS) is challenging due to their plethora of performance-related parameters and the complex interdependencies among subsets of these parameters [45, 64, 17]. For example, Cassandra has 56 performance tuning parameters and Redis has 46 such parameters. Several prior works like Rafiki [45], OtterTune [64], BestConfig [69], and others [17, 62, 61], have solved the problem of optimizing a DBMS when workload characteristics relevant to the data operations are relatively static. We call these "*static configuration tuners*". However, these solutions cannot decide on a set of configurations over a window of time in which the workloads are changing, i.e., what configuration to change to and when. Further, existing solutions cannot perform the reconfiguration of a cluster of database instances without degrading data availability.

Workload changes lead to new optimal configurations. However, it is not always desirable to switch to new configurations because the new workload pattern may be short-lived. Each reconfiguration action in clustered databases incurs costs because the server instance often needs to be restarted for the new configuration to take effect, causing a transient hit to performance during the reconfiguration period. In the case of dynamic workloads, the new workload may not last long enough for the reconfiguration cost to be recouped over a time window of interest to the system owner. Therefore, a proactive technique is required to estimate when executing a reconfiguration is going to be globally beneficial.

Fundamentally, this is where the drawback of all prior approaches to automatic performance tuning of DBMS lies—in the face of dynamic changes to the workload, they are either silent on when to reconfigure or perform a naïve reconfiguration whenever the workload changes. We show that a naïve reconfiguration, which is oblivious to the reconfiguration cost, actually *degrades* the performance for dynamic workloads relative to the default configurations and also relative to the best static configuration achieved using a static tuner with historical data from the system (Figure 3). For example, during periods of high dynamism in the read-write switches in a metagenomics workload in the largest metagenomics portal called MG-RAST [50], naïve reconfiguration degrades throughput by a substantial 61.8% over default.

Our Solution: We develop an online reconfiguration system—SOPHIA—for a NoSQL cluster comprising of multiple server instances, which is applicable to dynamic workloads with various rates of workload shifts. SOPHIA uses historical traces of the workload to train a workload predictor, which is used at runtime to predict future workload

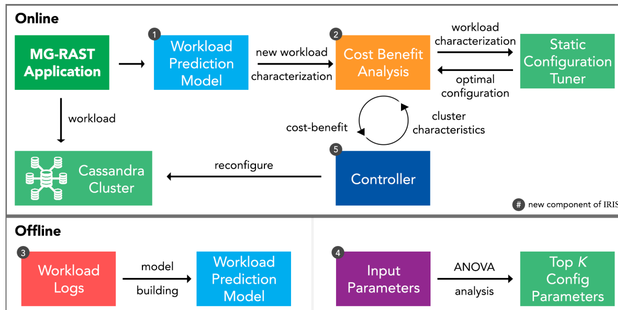


Figure 1: Workflow of SOPHIA *with offline model building and the online operation, plus the new components of our system. It also shows the interactions with the NoSQL cluster and a static configuration tuner, which comes from prior work.*

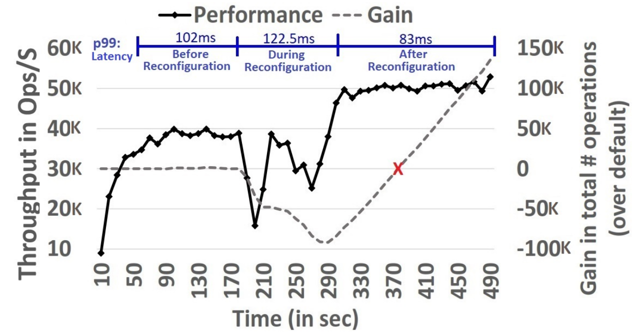


Figure 2: The effect of reconfiguration on the performance of the system. SOPHIA *uses the workload duration information to estimate the cost and benefit of each reconfiguration step and generates plans that are globally beneficial.*

patterns. Workload prediction is a challenging problem and has been studied in many prior works [43, 19, 51]. However, the workload predictor itself is not a contribution of SOPHIA, and it can operate with any workload predictor with sufficiently accurate and long-horizon predictions. SOPHIA searches the vast space of all possible reconfiguration plans, and hence determines the best plan through a novel Cost-Benefit-Analysis (CBA) scheme. For each shift in the predicted workload trace, SOPHIA interacts with any existing static configuration tuner (we use RAFIKI in our work because it is already engineered for NoSQL databases and is publicly available [15]), to quickly provide the optimal *point configurations* for the new workload and the estimated benefit from this new configuration. SOPHIA performs the CBA analysis, taking into account the predicted duration of the new workload and the estimated benefit from each reconfiguration step. Finally, for each reconfiguration step in the selected plan, SOPHIA initiates a distributed protocol to reconfigure the cluster without degrading data availability and maintaining the required data consistency requirement.

During its reconfiguration, SOPHIA can deal with different replication factors (RF) and consistency level (CL) requirements specified by the user. It ensures that the data remains continuously available through the reconfiguration process, with the required CL. This is done by controlling the number of server instances that are concurrently reconfigured. However, this is only possible when RF>CL. In cases where RF=CL, reconfiguring any node in the cluster will degrade data availability as every request will require a response from every replica before it is returned to the user. Therefore, we also implement an aggressive variant of our system (SOPHIA-AGGRESSIVE), which relaxes the data availability requirement in exchange for faster reconfiguration and hence better performance.

Evaluation Cases

We evaluate SOPHIA on two NoSQL databases, Cassandra [39] and Redis [7]. The first use case is based on real workload traces from the metagenomics analysis pipeline, MG-RAST [9, 49]. It is a global-scale metagenomics portal, the largest of its kind, which allows users to simultaneously upload their metagenomic data and use its computational pipeline. The workload does not have any discernible daily or weekly pattern, as the requests come from all across the globe and we find that the workload can change drastically over a few minutes. This presents a challenging use case as only 5 minutes or less of accurate lookahead is possible. The second use case is a bus-tracking application where read, scan, insert, and update operations are submitted to a backend database. The data has strong daily and weekly patterns to it. The third use case is a queue of data analytics jobs such as would be submitted to an HPC computing cluster. Here the workload can be predicted over long time horizons (order of an hour) by observing the jobs in the queue and leveraging the fact that a significant fraction of the job patterns are recurring. Thus, our three cases span the range of patterns and corresponding predictability of the workloads.

We compare our approach to existing solutions and show that SOPHIA increases throughput (and decreases tail-latency) under all dynamic workload patterns and for all types of queries, with no downtime. For example, SOPHIA achieves 24.5% higher throughput over default configurations and 21.5% higher throughput over a statically determined idealized optimal configuration in the bus-tracking workload. SOPHIA achieves 38% and 30% higher aggregate throughput over these two baselines in the HPC cluster workload. With SOPHIA's auto-tuning capability, Redis is able to operate through the entire range of workload sizes and read/write intensities, while the *vanilla Redis fails with large workloads*. The main contributions of SOPHIA are:

1. We show that state-of-the-art static tuners when applied to dynamic workloads degrade throughput below the state-of-practice of using the default parameter values and also degrade data availability.

2. SOPHIA performs cost-benefit analysis to achieve *long-horizon optimized performance for clustered NoSQL instances* in the face of dynamic workload changes, including unpredictable and fast changes to the workload.

3. SOPHIA executes a decentralized protocol to gracefully

switch over the cluster to the new configuration while respecting the data consistency guarantees and keeping data continuously available to users.
First, we show the improvement of using SOPHIA to tune a Cassandra cluster. Afterwards, we show how SOPHIA can be used to tune Redis and improve its performance. The rest of the paper is organized as follows. Section 2 provides an overview of our solution SOPHIA. We provide a background on Cassandra and its sensitivity to configuration parameters and on static configuration tuners in Section 3. We describe our solution in Section 4. We provide details of the workloads and our implementation in Section 5. We give the evaluation results in Section 6 and finally conclude.

2 Overview of SOPHIA

Here we give an overview of the workflow and the main components of SOPHIA. A schematic of the system is shown in Fig. 1. Details of each component are in Sec. 4.

SOPHIA runs as a separate entity outside the Cassandra cluster. It measures the workload by intercepting and observing received queries at the entry point(s) to Cassandra. Periodically, SOPHIA queries the **Workload Predictor** (box 1 in figure) to determine if any future workload changes exist that may merit a reconfiguration—no change also contributes information for the SOPHIA algorithm. Also, an event-driven trigger occurs when the predictor identifies a workload change. The prediction model is initially trained from representative workload traces from prior runs of the application and incrementally updated with additional data as SOPHIA operates. With the predicted workload, SOPHIA queries a static configuration tuner that provides the optimal configuration for a single point in time in the predicted workload. The static configuration tuner is initially trained on the same traces from the system as the workload predictor. Similarly, the static configuration tuner is also trained incrementally like the workload predictor.
The **Dynamic Configuration Optimizer** (box 2) generates a time-varying reconfiguration plan for a given look-ahead window using cost-benefit analysis (CBA). This plan gives both the time points when reconfiguration should be initiated *and* the new configuration parameters at each such time point. The CBA considers both the static, point solution information and the estimated, time-varying workload information. It is run every look-ahead time window apart or when the workload characteristics have changed significantly enough. The **Controller** (box 3) initiates a distributed protocol to gracefully switch the cluster to new configurations in the reconfiguration plan (Sec.4.5). This controller is conceptually centralized but replicated and distributed in implementation using off-the-shelf tools like ZooKeeper. SOPHIA decides how many instances to switch concurrently such that the cluster always satisfies the user's availability and consistency requirements. The Workload Predictor is located at a point where it can observe the aggregate workload such as at a gateway to the database cluster or by querying each database instance for its near past workload profile. The Dynamic Configuration Optimizer runs at a dedicated node close to the workload monitor. A distributed component runs on each node to apply the new reconfiguration plan.
Cost-Benefit Analysis in the Reconfiguration Plan
Each reconfiguration has a cost, due to changing parameters that require restarting or otherwise degrading the database services, e.g., by flushing the cache. The CBA in SOPHIA calculates the costs of implementing a reconfiguration plan by determining the number, duration, and magnitude of degradations. If a reconfiguration plan is found globally beneficial, the controller initiates the plan, else it is rejected. This insight, and the resulting protocol design to decide *whether* and *when* to reconfigure, are the fundamental contributions of SOPHIA.
Now we give a specific example of this cost-benefit trade-off from real MG-RAST workload traces. Consider the example in Fig. 2 where we apply SOPHIA's reconfiguration plan to a cluster of 2 servers with an availability requirement that at least 1 of 2 be online (i.e. CL=1). The Cassandra cluster starts with a read-heavy workload but with a configuration C_1 (Cassandra's default), which favors a write-heavy workload and is therefore suboptimal. With this configuration, the cluster provides a throughput of $\sim$40,000 ops/s and a tail latency of 102 ms (P99), but a better read-optimized configuration C_2 exists, providing $\sim$50,000 ops/s and a tail latency of 83 ms. The Cassandra cluster is reconfigured to the new C_2 configuration setting, using SOPHIA's controller, resulting in a temporary throughput loss due to the transient unavailability of the server instances as they undergo the reconfiguration, one instance at a time given the specified availability requirement. Also during the reconfiguration period, the tail latency increases to 122.5 ms on average. The two dips in throughput at 200 and 270 seconds correspond to the two server instances being reconfigured serially, in which two spikes in tail latency of 180 ms are observed. We plot, using the dashed line, the gain (benefit minus cost) over time in terms of the total # operations served relative to the default configuration. We see that there is a crossover point (the red X point) with the duration of the new workload pattern. If the predicted workload pattern lasts longer than this threshold (190 seconds from the beginning of reconfiguration in our example), then there is a gain from this step and SOPHIA would include it in the plan. Otherwise, the cost will outweigh the benefit, and any solution implemented without the CBA risks degrading the overall system performance. *Thus, a naïve solution (a simple extension of all existing static configuration tuners) that always reconfigures to the best configuration for the current workload will actually degrade performance for any reasonably fast-changing workload.* Therefore, a workload predictor and a cost-benefit analysis model are needed to develop a reconfiguration plan that achieves globally optimal performance over time.

3 Background

Overview of Apache Cassandra: Cassandra is one of the most popular NoSQL databases that is being used by many companies such as Apple, eBay, Netflix and many others [8]. This is because of its durability, scalability, and fault-tolerance, which are essential features for production deployments with large volumes of data. To be able to support a wide range of applications and access patterns, Cassandra (like many other DBMS) exposes many configuration parameters that control its internal behavior and affect its performance. This is intended to customize the DBMS for widely different applications. According to the Cassandra architecture, it caches writes in an in-memory Log-structured merge tree [58] called *Memtable* for a certain period of time. Afterwards, all Memtables get flushed to their corresponding persistent representation on disk called *SSTables*. The same flushing process can be triggered if the size of the Memtable exceeds a specific threshold. A Memtable is always flushed to a new SSTable, which is never updated after construction. Consequently, a single key can be stored accross many SSTables with different timestamps, and therefore a read request to that key will require Cassandra to scan through all existing SSTables and retrieve the one with the latest timestamp. To keep the number of SSTables manageable, Cassandra applies a compaction strategy, which combines a number of old SSTables into one while removing obsolete records. This achieves better performance for reads, but is also a heavy operation that consumes CPU and memory and can negatively impact the performance for writes during compaction.

Dynamic Workloads in Cassandra: Optimal values of these performance-sensitive parameters are dependent on the workload. For example, we find empirically that size-tiered compaction strategy achieves 44% better performance for write-heavy workloads than leveled compaction strategy, while leveled compaction strategy achieves 90% better performance for read-heavy workloads (Figure 3). When the workload changes, the optimal parameters for the new workload will likely change as well. An incremental approach is desired, rather than restarting all servers concurrently, which renders all the data unavailable during reconfiguration.

Workloads in our pipeline have shifts in the number of requests/s and also the relative ratio of the different operations on the database (i.e., transaction mixture). Therefore, SOPHIA needs to react in an agile manner to such shifts. For example, MG-RAST traces show 443 sharp (more than 80% change) shift/day on average, mostly from read-heavy to write-heavy workloads and vice-versa. For the bus-tracking application, a smaller, but still significant, value of 63 shift/-day is observed. The static tuners cannot handle such dynamism and cannot even pick a single parameter set that will on an average give the highest throughput aggregated over a window of time because of the lack of lookahead and also the lack of the Cost-Benefit analysis model.

4 Design of SOPHIA

SOPHIA seeks to answer the following two broad questions: When should the cluster be reconfigured? How should we apply the reconfiguration steps? The answer to the first question leads to what we call a *reconfiguration plan*. The answer to the second question is given by our distributed protocol that reconfigures the various server instances in rounds. Next, we describe SOPHIA's components.

4.1 Workload Modeling and Forecasting: In a generic sense, we can define the workload at a particular point in time as a vector of various time-varying features:

$$\boldsymbol{W}(t) = \{p_1(t), p_2(t), p_3(t), ..., p_n(t)\} \qquad (1)$$

where the workload at time t is $\boldsymbol{W}(t)$ and $p_i(t)$ is the time-varying i-th feature. These features may be directly measured from the database, such as the load (i.e., requests/s) and the occupancy level of the database, or they may come from the computing environment, such as the number of users or jobs in a batch queue. These features are application dependent and are identified by analyzing the application's historical traces. Details for time-varying features of each application are described in Section 5. For workload forecasting, we discretized time into sliced T_d durations ($= 30s$ in our model) to bound the memory and compute cost. We then predicted future workloads as:

$$\boldsymbol{W}(t_{k+1}) = f_{\text{pred}}(\boldsymbol{W}(t_k), \boldsymbol{W}(t_{k-1}), ..., \boldsymbol{W}(t_0)) \qquad (2)$$

where k is the current time index into T_d-wide steps. For ease of exposition for the rest of the paper, we drop the term T_d, assuming implicitly that this is one time unit. The function f_{pred} is any function that can make such a prediction, and in SOPHIA, we utilize a simple Markov-Chain model for MG-RAST and Bus-Tracking, while we use a deterministic, fully accurate output from a batch scheduler for the HPC data analytics workload, i.e., a perfect f_{pred}. However, more sophisticated estimators, such as neural networks [43, 31, 33], even with some degree of interpretability [32], have been used in other contexts and SOPHIA is modular enough to use any such predictor.

4.2 Adapting a Static Configuration Tuner for SOPHIA: SOPHIA uses a static configuration tuner (RAFIKI), designed to work with Cassandra, to output the best configuration for the workload at any given point in time. RAFIKI uses Analysis-of-variance (ANOVA) [55] in order to estimate the importance of each parameter. It selects the top-k parameters in its configuration optimization method, which is in turn determined by a significant drop-off in the importance score. The ability to adapt optimized "kernels" to build robust algorithms comes from our vision to accelerate the pipeline of creating efficient algorithms, conceptualized in Sarvavid [44]. The 7 highest performance-sensitive parameters for all three of our workloads are: (1) Compaction method, (2) # Memtable flush

writers, (3) Memory-table clean-up threshold, (4) Trickle fsync, (5) Row cache size, (6) Number of concurrent writers, and (7) Memory heap space. These parameters vary with respect to the reconfiguration cost that they entail. The change to the compaction method incurs the highest cost as it causes every Cassandra instance to read all its SSTables and re-write them to the disk in the new format. However, due to inter-dependability between these parameters, the compaction frequency is still being controlled by reconfiguring the second and third parameters with the cost of a server restart. Similarly, parameters 4, 6, 7 need a server restart for their new values to take effect and these cause the next highest level of cost. Finally, some parameters (parameter 5 in our set) can be reconfigured without needing a server restart and therefore have the least level of cost.

In general, the database system has a set of performance-impactful configuration parameters $\boldsymbol{C} = \{c_1, c_2, \cdots, c_n\}$ and the optimal configuration $\boldsymbol{C}_{opt}$ depends on the particular workload $\boldsymbol{W}(t)$ executing at that point in time. In order to optimize performance across time, SOPHIA needs the static tuner to provide an estimate of throughput for both the optimal and the current configuration for any workload:

$$H_{\text{sys}} = f_{\text{ops}}(\boldsymbol{W}(t), \boldsymbol{C}_{\text{sys}}) \tag{3}$$

where H_{sys} is the throughput of the cluster of servers with a configuration $\boldsymbol{C}_{\text{sys}}$ and $f_{\text{ops}}(\boldsymbol{W}(t), \boldsymbol{C}_{\text{sys}})$ provides the system-level throughput estimate. $\boldsymbol{C}_{\text{sys}}$ has $N_s \times |\boldsymbol{C}|$ dimensions for N_s servers and C different configurations. Cassandra by careful design achieves efficient load balancing across multiple instances whereby each contributes approximately equally to the overall system throughput [39, 20]. Thus, we define a single server average performance as $H_i = \frac{H_{\text{sys}}}{N_s}$.

From these models of throughput, optimal configurations can be selected for a given workload:

$$C_{\text{opt}}(\boldsymbol{W}(t)) = \arg\max_{\boldsymbol{C}_{\text{sys}}} H_{\text{sys}} = \arg\max_{\boldsymbol{C}_{\text{sys}}} f_{\text{ops}}(\boldsymbol{W}(t), \boldsymbol{C}_{\text{sys}}) \tag{4}$$

In general, $\boldsymbol{C}_{\text{opt}}$ can be unique for each server, but in SOPHIA, it is the same across all servers and thus has a dimension of $|\boldsymbol{C}|$ making the problem computationally easier. This is due to the fact that SOPHIA makes a design simplification—it performs the reconfiguration of the cluster as an atomic operation. Thus, it does not abort a reconfiguration action mid-stream and all servers must be reconfigured in round i prior to beginning any reconfiguration of round $i+1$. We also speed up the prediction system f_{ops} by constructing a cached version with the optimal configuration $\boldsymbol{C}_{opt}$ for a subset of $\boldsymbol{W}$ and using nearest-neighbor lookups whenever a near enough neighbor is available.

4.3 Dynamic Configuration Optimization:

SOPHIA's core goal is to maximize the total throughput for a database system when faced with dynamic workloads. This introduces time-domain components into the optimal configuration strategy $\boldsymbol{C}_{\text{opt}}^T = \boldsymbol{C}_{\text{opt}}(\boldsymbol{W}(t))$, for all points in (discretized) time till a lookahead T_L. Here, we describe the mechanism that SOPHIA uses for CBA modeling to construct the best reconfiguration plan (defined formally in Eq. 5) for evolving workloads.

In general, finding solutions for $\boldsymbol{C}_{\text{opt}}^T$ can become impractical since the possible parameter space for $\boldsymbol{C}$ is large and the search space increases linearly with T_L. To estimate the size of the configuration space, consider that in our experiments we assumed a lookahead $T_L = 30$ minutes and used 7 different parameters, some of which are continuous, e.g., `Memory-table clean-up threshold`. If we take an underestimate of each parameter being binary, then the size of the search space becomes $2^{7\times 30} = 1.6 \times 10^{63}$ points, an impossibly large number for exhaustive search. We define a compact representation of the reconfiguration points (Δ's) to easily represent the configuration changes. The maximum number of switches within T_L, say M, is bounded since each switch takes a finite amount of time. The search space for the dynamic configuration optimization is then $Combination(TL, M), M) \times |\boldsymbol{C}|$. This comes from the fact that we have to choose at most M points to switch out of all the T_L time points and at each point there are $|\boldsymbol{C}|$ possible configurations. We define the reconfiguration plan as:

$$\boldsymbol{C}_{\text{sys}}^{\Delta} = [\boldsymbol{T} = \{t_1, t_2, ..., t_M\}, \boldsymbol{C} = \{C_1, C_2, ..., C_M\}] \tag{5}$$

where t_k is a point in time and C_k is the configuration to use at t_k. Thus, the reconfiguration plan gives *when* to perform a reconfiguration and at each such point, *what* configuration to choose.

The objective for SOPHIA is to select the best reconfiguration plan $(\boldsymbol{C}_{\text{sys}}^{\Delta})^{\text{opt}}$ for the period of optimization, lookahead time T_L:

$$(\boldsymbol{C}_{\text{sys}}^{\Delta})^{\text{opt}} = \arg\max_{\boldsymbol{C}_{\text{sys}}^{\Delta}} B(\boldsymbol{C}_{\text{sys}}^{\Delta}, \boldsymbol{W}) - L(\boldsymbol{C}_{\text{sys}}^{\Delta}, \boldsymbol{W}) \tag{6}$$

where $\boldsymbol{C}_{\text{sys}}^{\Delta}$ is the reconfiguration plan, B is the benefit function, and L is the cost (or loss) function, and $\boldsymbol{W}$ is the time-varying workload description. Detailed derivation of functions B and L is shown in Supplemental Material (Section 9.1). When SOPHIA will extend to allow scale out, we will have to consider the data movement volume as another cost to minimize. The L function captures the opportunity cost of having each of N_s servers offline for T_r seconds for the new workload versus if the servers remained online with the old configuration. As the node downtime due to reconfiguration never exceeds Cassandra's threshold for declaring a node is dead (3 hours by default), data-placement tokens are not re-assigned due to reconfiguration. Therefore, we do not include cost of data movement in functions L. SOPHIA can work with any reconfiguration cost, including different costs for different parameters—these can be fed into the loss function L.

The objective is to maximize the time-integrated gain (benefit – cost) of the reconfiguration from Eq. (6). The three

unknowns in the optimal plan are M, $\boldsymbol{T}$, and $\boldsymbol{C}$, from Eq. (5). If only R servers can be reconfigured at a time (explained in Sec. 4.5 how R is calculated), at least $T_r \times \frac{N_s}{R}$ time must elapse between two reconfigurations. This puts a limit on M, the maximum number of reconfigurations that can occur in the lookahead period T_L.

A greedy solution for Eq. (6) that picks the first configuration change with a net-increase in benefit may produce suboptimal $\boldsymbol{C}^{\Delta}_{\text{sys}}$ over the horizon T_L because it does not consider the coupling between multiple successive workloads. For example, considering a pairwise sequence of workloads, the best configuration may not be optimal for either $\boldsymbol{W}(t_1)$ or $\boldsymbol{W}(t_2)$ but *is* optimal for the paired sequence of the two workloads. This could happen if the same configuration gives reasonable performance for $\boldsymbol{W}(t_1)$ or $\boldsymbol{W}(t_2)$ and has the advantage that it does not have to switch during this sequence of workloads. This argument can be naturally extended to longer sequences of workloads.

A T_L value that is too long will cause SOPHIA to include decision points with high prediction errors, and a value that is too short will cause SOPHIA to make almost greedy decisions. The appropriate lookahead period is selected by benchmarking the non-monotonic but convex throughput while varying the lookahead duration and selecting the point with maximum end-to-end throughput. We give our choices for our three applications when describing the first experiment with each application (Section 6).

4.4 Finding Optimal Reconfiguration Plan with Genetic Algorithms: We use a heuristic search technique, Genetic Algorithms or GA, to find the optimal reconfiguration plan. Although meta-heuristics like GA do not guarantee finding global optima, they have two desirable properties for SOPHIA. Our space is non-convex because many of the parameters impact the same resources such as CPU, RAM, and disk, and settings of one parameter impact the others.Therefore, greedy or gradient descent-based searches are prone to converge to a local optima. Also the GA's tunable completion is needed in our case for speedy decisions, as the optimizer executes in the critical path.
The representation of the GA solution incorporates two parts. First, the chromosome orientation, which is simply the reconfiguration plan (Eq. 5). The second part is the fitness function definition used to assess the quality of different reconfiguration plans. For this, we use the cost-benefit analysis as shown in Eq. 6 where fitness is the total number of operations (normalized for bus-tracking traces to account for different operations' scales) for the T_L window for the tested reconfiguration plan and given workload. We build a simulator to apply the individual solutions and to collect the corresponding fitness values, which are used to select the best solutions and to generate new solutions in the next generation. We utilize a Python library, `pyeasyga`, with 0.8 crossover fraction and population size of 200. We run 10 concurrent searches and pick the best configuration from those. The runtime of this algorithm is dependent on the length of the lookahead period and the number of decision points. For MG-RAST, the GA has 30 decision points in the lookahead period and results in execution time of 30-40 sec. For the HPC workload, the number of decision points is 180 as it has a longer lookahead period, resulting in a runtime of 60-70 sec. For the bus-tracking workload, the GA has 48 decision points and a runtime of 20-25 sec. The GA is typically re-run toward the end of the lookahead period to generate the reconfiguration plan for the next lookahead time window. Also, if the actual workload is different from the predicted workload, the GA is re-invoked. This last case is rate limited to prevent too frequent invocations of the GA during (transient) periods of non-deterministic behavior of the workload.

4.5 Distributed Protocol for Online Reconfiguration: Cassandra and other distributed databases maintain high availability through configurable redundancy parameters, consistency level (CL) and replication factor (RF). CL controls how many confirmations are necessary for an operation to be considered successful. RF controls how many replicas of a record exist throughout the cluster. Thus, a natural constraint for each record is RF $\geq$ CL. SOPHIA queries token assignment information (where a token represents a range of hash values of the primary keys which the node is responsible for) from the cluster, using tools that ship with all popular NoSQL distributions (like `nodetool ring` for Cassandra), and hence constructs what we call a *minimum availability subset* (N_{minCL} for short). We define this subset as the minimum subset of nodes that covers at least CL replicas of *all* keys. To meet CL requirements, SOPHIA insures that N_{minCL} nodes are operational at any point of time. Therefore, SOPHIA makes the design decision to configure up to $R = N_s - N_{minCL}$ servers at a time, where N_{minCL} depends on RF, CL, and token assignment. If we assume a single token per node (Cassandra's default with vnodes disabled), then a subset of $\lceil \frac{N_s}{RF} \rceil$ nodes covers all keys at least once. Consequently, N_{minCL} becomes CL $\times \lceil \frac{N_s}{RF} \rceil$ to cover all keys at least CL times. Thus, the number of reconfiguration steps $= \frac{N_s}{R} = \frac{RF}{RF-CL}$ becomes independent of the cluster size N_s.
In the case where RF = CL, N_{minCL} becomes equivalent to N_s and hence SOPHIA cannot reconfigure the system, without harming data availability, hence we use the SOPHIA-AGGRESSIVE variant in that case. However, we expect most systems with high consistency requirements to follow a read/write quorum with CL $= \lceil \frac{\text{RF}}{2} \rceil$ [23]. Note that SOPHIA reduces the number of available data replicas during the transient reconfiguration periods, and hence reduces the system's resilience to additional failures. However, one optional parameter in SOPHIA is how many failures during reconfiguration the user will want to tolerate (our experiments were run with zero). This is a high-level

parameter that is intuitive to set by the database admin. Also notice that data that existed on the offline servers prior to reconfiguration is not lost due to the drain step, but data written during the transient phase has lower redundancy until the reconfigured servers get back online. In order to reconfigure a Cassandra cluster, SOPHIA performs the following steps, R server instances at a time:

51. Drain: Before shutting down a Cassandra instance, we flush the entire Memtable to disk by using Cassandra's tool `nodetool drain` and this ensures that there are no pending commit logs to replay upon a restart.

1. **Drain**: Before shutting down a Cassandra instance, we flush the entire Memtable to disk by using Cassandra's tool `nodetool drain` and this ensures that there are no pending commit logs to replay upon a restart.
2. **Shutdown**: The Cassandra process is killed on the node.
3. **Configuration file**: Replace the configuration file with new values for all parameters that need changing.
4. **Restart**: Restart the Cassandra process on the same node.
5. **Sync**: SOPHIA waits for Cassandra's instance to completely rejoin the cluster by letting a coordinator know of where to locate the node and then synchronizing the missed updates during the node's downtime.

In Cassandra, writes for down nodes are cached by available nodes for some period and re-sent to the nodes when they rejoin the cluster. The time that it takes to complete all these steps for one server is denoted by T_r, and T_R for the whole cluster, where $T_R = T_r \times \frac{RF}{RF-CL}$. During all steps 1-5, additional load is placed on the non-reconfiguring servers as they must handle the additional write and read traffic. Step 5 is the most expensive and typically takes 60-70% of the total reconfiguration time, depending on the amount of cached writes. We minimize step 4 practically by installing binaries from the RAM and relying on draining rather than commit-log replaying in step 1, reducing pressure on the disk.

5 Datasets

MG-RAST Workload: We use real workload traces from MG-RAST, the leading metagenomics portal operated by the US Department of Energy. As the amount of data stored by MG-RAST has increased beyond the limits of traditional SQL stores (23 PB as of August 2018), it relies on a distributed NoSQL Cassandra database cluster. Users of MG-RAST are allowed to upload "jobs" to its pipeline, with metadata to annotate job descriptions. All jobs are submitted to a computational queue of the US Department of Energy private Magellan cloud. We analyzed 80 days of query trace from the MG-RAST system from April 19, 2017 till July 9, 2017. From this data, we make several observations: (i) Workloads' read ratio (RR) switches rapidly with over 26,000 switches in the analyzed period. (ii) A negative correlation of -0.72 is observed between the Workloads' read ratio and number of requests/s (i.e., load). That is due to the fact that most of the write operations are batched to improve network utilization. (iii) Majority (i.e., more than 80%) of the switches are abrupt, from RR=0 to RR=1 or vice versa. (iv) KRD (key reuse distance) is very large. (v) No daily or weekly workload pattern is discernible, as expected for a globally used cyberinfrastructure.

Bus Tracking Application Workload: Secondly, we use real workload traces from a bus-tracking mobile application called **Tiramisu** [43]. The system provides live tracking of the public transit bus system. It updates bus locations periodically and allows users to search for nearby bus stops. There are four types of queries—read, update, insert, and scan (retrieving all the records in the database that satisfy a given predicate, which is much heavier than the other operations). A sample of the traces is publicly available [42]. We trained our model using 40 days of query traces, while 18 days were used as testing data. We draw several observations from this data: (i) The traces show a daily pattern of workload switches. For example, the workload switches to scan-heavy in the night and switches to update-heavy in the early morning. (ii) The Workload is a mixture of Update, Scan, Insert, and Read operations in the ratio of 42.2%, 54.8%, 2.82%, and 0.18% respectively. (iii) KRD is very small. From these observations, we notice that the workload is very distinct from MG-RAST and thus provides a suitable point for comparison.

Simulated Analytics Workload: For long-horizon reconfiguration plans, we simulate synthetic workloads representative of batch data analytics jobs, submitted to a shared HPC queue. We integrate SOPHIA with a job scheduler (like PBS [27]), that examines jobs while they wait in a queue prior to execution. Thus, the scheduler can profile the jobs waiting in the queue, and hence forecast the aggregate workload over a lookahead horizon, which is equal to the length of the queue. We model the jobs on data analytics jobs submitted to a Microsoft Cosmos cluster [21] and as in that paper, we achieve high accuracy in predicting when a job will start executing. Thus, SOPHIA is able to drive long-horizon reconfiguration plans. Each job is divided into phases: a write-heavy phase resembling an upload phase of new data, a read-heavy phase resembling executing analytical queries to the cluster, and a third, write-heavy phase akin to committing the analysis results. However, some jobs can be recurring (as shown in [1, 21]) and running against already uploaded data. These jobs will execute the analysis phase directly, skipping the first phase. The size of each phase is a random variable with U(200,100K) operations, and whenever a job finishes, a new job is selected from the queue and executed. We vary the level of concurrency and have an equal mix of the

two types of jobs and monitor the aggregate workload. Figure 11 in Supplemental Material shows the synthetic traces for three job sizes. With increase in concurrency, the aggregate pattern becomes smoother and the latency of individual jobs increases.

6 Experimental Results

Here we evaluate the performance of SOPHIA under different experimental conditions for the 3 applications. We use a simple query model typical for NoSQL databases and is in contrast to complex analytics queries supported by more complex database engines. Hence, our throughput is defined as the number of queries per second. In all experiments, we collect both throughput and tail latency (p99) performance metrics. However, since the two parameters have an almost perfect inverse relationship in all experiments, we omit tail latency (except in Figures 4 and 6). We evaluate SOPHIA on Amazon EC2 using instances of size M4.xlarge with 4 vCPU's, 16 GB of RAM, provisioned IOPS (SSD) EBS for storage and network bandwidth of 0.74 Gbits/s for all Cassandra servers and workload drivers. Each node is loaded with 6 GB of data initially (SOPHIA's performance is evaluated with greater data volumes in Experiment 4). We use multiple concurrent clients to saturate the database servers and aggregate the throughput and tail latency observed by every client.

Baseline Comparisons

We compare the performance of SOPHIA to baseline configurations (1-5). We consider 3 variants of SOPHIA (6-8).

(1) Default: The default configuration that ships with Cassandra. This configuration favors write-heavy workloads by design [48].

(2) Static Optimized: This baseline resembles the static tuner (RAFIKI) when queried to provide the one *constant* configuration that optimizes for the entire future workload. This is an impractically ideal solution since it is assumed here that the future workload is known perfectly. However, non-ideally no configuration changes are allowed dynamically.

(3) Naïve Reconfiguration: Here, when the workload changes, RAFIKI's provided reconfiguration is always applied, instantiated by concurrently shutting down all server instances, changing their configuration parameters, and restarting all of them. Practically, this makes data unavailable and may not be tolerable in many deployments such as user-facing applications. The static configuration tuners are silent about when the optimal configurations determined by them must be applied and this baseline is a logical instantiation of all of the prior work.

(4) ScyllaDB: The performance of NoSQL database ScyllaDB [57] in its vanilla form. ScyllaDB is touted to be a much faster (10X or higher) drop-in replacement to Cassandra [56]. This stands in for other self-tuning databases [30].

(5) Theoretical Best: This baseline resembles the theoretically best achievable performance over the predicted workload period. This is simulated by assuming Cassandra is running with the optimal configuration at any point of time and not penalizing it for the cost of reconfiguration. This serves as an upper bound for the performance.

(6) SOPHIA with Oracle: This is SOPHIA with a fully accurate workload predictor.

(7) SOPHIA-AGGRESSIVE: A variant from SOPHIA that prefers faster reconfiguration over data availability and is used only when RF=CL. SOPHIA-AGGRESSIVE violates the availability requirement by reconfiguring all servers at the same time. Unlike Naïve, it uses the CBA model to decide when to reconfigure, and therefore it does not execute reconfiguration every time the workload changes.

(8) SOPHIA: This is our complete system.

Major Insights

We draw some key insights from the experimental results. First, globally shared infrastructures with black-box jobs only allow for short-horizon workload predictions. This causes SOPHIA to take single-step reconfiguration plans and limits its benefit over a static optimized approach (Figure 3). In contrast, when job characteristics can be predicted well (bus tracking and data analytics applications), SOPHIA achieves significant benefit over both default and static optimized cases (Figures 4 and 5). This benefit stays even when there is significant uncertainty in predicting the exact job characteristics as shown in Figure 9. Second, Cassandra can be used in preference to the recent popular drop-in ScyllaDB, an auto-tuning database, with higher throughput across the entire range of workload types, as long as we overlay a dynamic tuner, such as SOPHIA, atop Cassandra (Figures 3 and 5). Third, as the replication factor increases while the number of server are fixed, the reconfiguration time of SOPHIA decreases, thus improving its benefit (Figure 7). Contrarily, as CL increases, the benefit of SOPHIA shrinks (Figure 7).Finally, SOPHIA is applied to a different NoSQL database, Redis, and solves a long-standing configuration problem with it, one which has caused Redis to narrow its scope to being an in-memory database only (Figure 10).

Experiment 1: MG-RAST Workload

We present our experimental evaluation with 20 test days of MG-RAST data. To zoom into the effect of SOPHIA with different levels of dynamism in the workload, we segment the workload into 4 scenarios and present those results in addition to the aggregated ones.

Workload Prediction Model: We created 16,512 training samples composed of $T_d = 5min$ steps across the 60 days MG-RAST workloads. We compare the performance of a first-order and a second-order Markov Chain model. We represent the states as the proportion of read operations during the T_d interval. We use a quantization level of 10% in the read ratio between different states. We categorize

the test days into 4: “Slow”, “Medium”, and “Fast”, by the frequency of switching from the read- to the write-intensive workloads and this maps to the average read ratios (RR) shown in Table 1. “Write” represents days with long write-heavy periods. Table 1 shows the prediction RMSE for the four representative workload scenarios. Because of the lack of application-level knowledge, in addition to the well-known uncertainty in job execution times in genomics pipelines [40], the Markov Chain model only provides accurate predictions for short time intervals. Moreover, increasing the order of the model has very little impact on the prediction performance and also increases the number of states (11 states in the First-order model vs. 120 states in the Second-order model). We also tried to train a more complex model (RNN) but its prediction quality was similar. We notice that the best accuracy is for the “Slow” scenario, whereas it drops below 50% for “Medium”, and it is always below 50% for the “Fast” and “Write” scenarios. Because the “Slow” scenario is the most common (observed 74% of time in the training data), we use a value of $T_L = 5min$.

Table 1: RMSE for predicting MG-RAST and Bus-Tracking workloads.

MG-RAST					
MC-Order	First		Second		RR
Frequency	5m	10m	5m	10m	-
Slow	34.4%	56%	34%	55%	70%
Medium	59%	90%	59%	89%	50%
Fast	66%	93%	63%	89%	45%
Write	52.8%	76.1%	51.5%	75.5%	35%
Aggregate	43.7%	68.7%	43.4%	68.2%	-

Bus-Tracking		
Lookahead	First	Second
15m	6.9%	7.12%
1h	7.4%	7.4%
2h	7.9%	7.4%
5h	10%	7.5%
10h	13.7%	8%
#States	117	647

Performance Comparison:

Now we show the performance of SOPHIA with respect to the four workload categories. We first present the result with the smallest possible number of server instances, 4, run with operational MG-RAST’s parameters RF=3 and CL=1 [35]. We show the result in terms of total operations for each test workload as well as a weighted average “combined” representation that models behavior for the entire MG-RAST workload. Figure 3 shows the performance improvements for our test cases.

From Figure 3, we see that SOPHIA always outperforms naïve in total ops/s (average of 31.4%) and individually in read (31.1%) and write (33.5%) ops/s. SOPHIA also outperforms the default for the slow and the mid frequency cases, while it slightly under performs in the fast frequency case. The average improvement due to SOPHIA across the 4 categories is 20.4%. The underperformance for the fast case is due to increased prediction error. Naïve baseline has a significant loss compared to default: 21.6%. The static optimized configuration (which for this workload favors read-heavy pattern) has a slightly higher throughput over SOPHIA by 6.3%. This is because the majority of the selected samples are read periods (RR=1), which hides the gain that SOPHIA achieves for write periods. However, we see that with respect to write operations, SOPHIA achieves 17.6% higher throughput than the static optimized configuration. Increased write

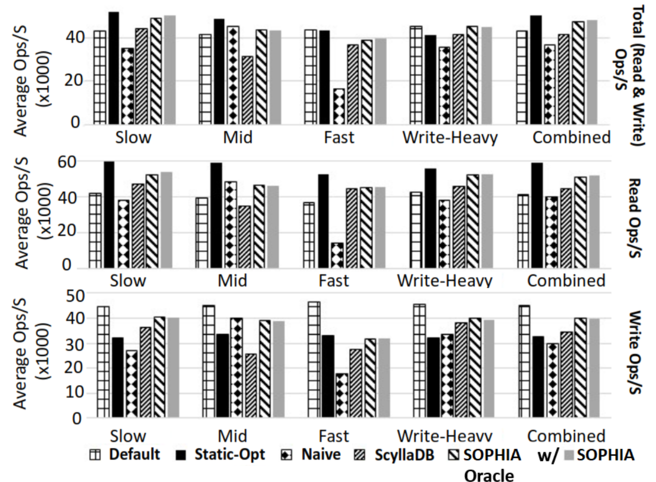


Figure 3: Improvement for four different 30-minute test windows from MG-RAST real traces over the baseline solutions.

throughput is critical for MG-RAST to support the bursts of intense writes. This avoids unacceptable queuing of writes, which can create bottlenecks for subsequent jobs that rely on the written shared dataset. Moreover, we observe that SOPHIA performs within 2-3% to SOPHIA w/ Oracle in all scenarios, which shows the minor impact of the workload predictor accuracy. For instance, SOPHIA w/ Oracle shows a 2% reduction in performance compared to SOPHIA in the slow trace. This is because Oracle has perfectly accurate predictions for $T_L = 5min$ only. With this very short lookahead, SOPHIA makes greedy reconfiguration decisions, and hence does not achieve globally optimal performance over other baselines.

ScyllaDB has an auto-tuning feature that is supposed to continuously react to changes in workload characteristics and the current state (such as, the amount of dirty memory state). ScyllaDB is claimed by its developers to outperform Cassandra in all workload mixes by an impressive 10X [56]. However, this claim is not borne out here and only in the read-heavy case (the “Slow” scenario) does ScyllaDB outperform. Even in this case, SOPHIA is able to reconfigure Cassandra at runtime and turn out a performance benefit over ScyllaDB. We conclude that based on this workload and setup, a system owner can afford to use Cassandra with SOPHIA for the *entire range* of workload mixes and not have to transition to ScyllaDB.

Experiment 2: Tiramisu Workload

We evaluate the performance of SOPHIA using the bus-tracking application traces. Figure 4 shows the gain of using SOPHIA over the various baselines. In this experiment, we report the normalized average Ops/s instead of the absolute average Ops/s metric. This means we normalize each of the 4 operation’s throughputs by dividing by the maximum Ops/s seen under a wide variety of configurations and then average the 4 normalized throughputs. The reason for

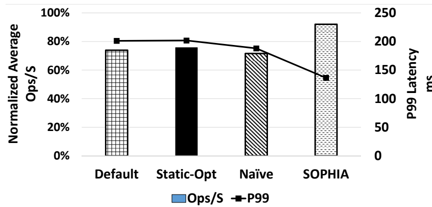


Figure 4: Gain of applying SOPHIA *to the bus-tracking application. We use 8 Cassandra servers with RF=3, CL=1. A 100% on the Y-axis represents the theoretical best performance.* SOPHIA *achieves improvements of 24.5% over default, 21.5% over Static-Opt, and 28.5% over naïve.*

Figure 5: Improvement for HPC data analytics workload with different levels of concurrency. We notice that SOPHIA *achieves higher average throughput over all baselines*

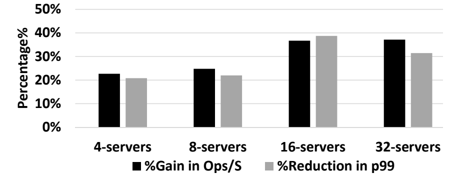


Figure 6: Improvement with scale using HPC workload with 5 jobs with RF=3 and CL=1. SOPHIA *provides consistent gains across scale because the cost of reconfiguration does not change with scale (for the same RF and CL). The higher gains for 16 and 32 servers is due to the use of M5 instances, which can be exploited by* SOPHIA *better than Static-Opt.*

this is that for this workload, different operations have vastly different throughput scales. For example, when the workload switches to a Scan-heavy phase, the performance of the cluster varies from 34 Ops/s to 219 Ops/s depending on the configuration of the cluster. For an Update-heavy phase, the performance varies from 1,739 Ops/s to 5,131 Ops/s. This is because Scan is a much heavier operation for the DBMS compared to Update.

SOPHIA outperforms default configuration by 24.5%, Static-Opt by 21.5%, and Naïve by 28.5%. The gains are higher because SOPHIA can afford longer lookahead times with accurate workload prediction. We notice that Naïve is achieving a comparable performance to both Default and Static-Opt configurations, unlike MG-RAST. This is because the frequency of workload changes is lower here. However, Naïve still renders the data unavailable during the reconfiguration period.

Workload Prediction Model: Unlike MG-RAST, the bus-tracking application traces show a daily pattern which allows our prediction model to provide longer lookahead periods with high accuracy (Table 1). We use a Markov Chain prediction model to capture the workload switching behavior. We start by defining the state of the workload as the proportion of each operation type in an aggregation interval (15 minutes in our experiments). For example, Update=40%, Read=20%, Scan=40%, Insert=0% represents a state of the workload. We use a quantization level of 10% in any of the 4 dimensions to define the state. We use the second-order Markov Chain with a lookahead period of 5 hours as this is when our prediction error is $\leq$ 8%. As expected theoretically, the second order model is more accurate at all lookahead times, since there is enough training data available for training the models. Seeing the seeming regular diurnal and weekly pattern in the workload, we create two simple predictor straw-men that uses only the current time-stamp or the current time-stamp and day of the week as input features to perform prediction. The predicted workload is the average of the mixtures at the previous 10 points. These predictors have unacceptably high RMSE of 31.4% and 24.0%. Therefore, although the workload is showing a pattern, we cannot generate the optimal plan once and use it for all subsequent days. Therefore, online workload monitoring and prediction is needed to achieve the best performance

Experiment 3: HPC Data Analytics

We evaluate the performance of SOPHIA using HPC data analytics workload patterns described in Section 5. Here our lookahead is the size of the job queue, which is conservatively taken as 1 hour. Figure 5 shows the result for the three levels of concurrency (1, 5, and 10 jobs). We see that SOPHIA outperforms the default for all the three cases, with average improvement of 30%. In comparison with Static-Opt (which is a different configuration in each of the three cases), we note that SOPHIA outperforms for the 1 job and 5 jobs cases by 18.9% and 25.7%, while it is identical for the 10 jobs case. This is because in the 10 jobs case, the majority of the workload lies between RR=0.55 and RR=0.85, and in this case, SOPHIA switches only once: from the default configuration to the same configuration as Static-Opt. We notice that SOPHIA achieves within 9.5% of the theoretical best performance for all three cases. We notice that SOPHIA achieves significantly better performance over Naïve by 27%, 13%, and 122% for the three cases. Naïve, in fact, degrades the performance by 32.9% (10 concurrent jobs). In comparison with ScyllaDB, SOPHIA achieves a performance benefit of 17.4% on average, which leads to a similar conclusion as in MG-RAST about the continued use of Cassandra.

Experiment 4: Scale-Out & Greater Volume

Figure 6 shows the behavior of SOPHIA with increasing scale using the data analytics workload. We show the comparison between SOPHIA and Static-Opt (all other baselines performed worse than Static-Opt). We use a weak scaling pattern, i.e., keeping the amount of data per server fixed while still operating close to saturation. We increase the number of shooters as well to keep the request rate per server fixed. By our design (Sec. 4.5), the number of reconfiguration steps stays constant with scale. We notice that the network bandwidth needed by Cassandra's gossip protocol increases with the scale of the cluster, causing the

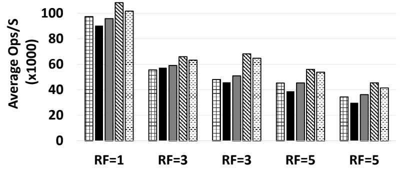


Figure 7: Effect of varying RF and CL on system throughput. We use a cluster of 8 nodes and compare the performance of SOPHIA *to Default, Static-Opt, and naïve.* SOPHIA *outperforms the static baselines and approaches the theoretical best as RF–CL increases.*

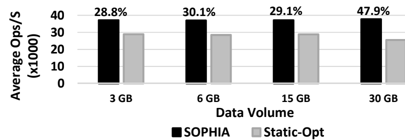


Figure 8: Effect on increasing data volume per node. We use a cluster of 4 servers and compare the performance to the static optimized. The results show that SOPHIA*'s gain is consistent with increasing data volumes per node.*

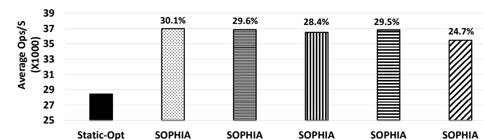


Figure 9: Effect of noise in workload prediction on the performance of SOPHIA *on the data analytics workload with level of concurrency = 5. The percentage represents the amount of noise added to the predicted workload pattern.*

network to become the bottleneck in the case of 16 and 32 servers when M4.xlarge instances are used. Therefore, we change the instance type to M5.xlarge in these cases (network bandwidth of 10 Gbit/s compared to 0.74 Gbit/s). The results show that SOPHIA's optimal reconfiguration policy has a higher performance over Static-Opt across all scales. Moreover, we see a higher gain in the cases of 16 and 32 servers since M5 instances have higher CPU power than M4 ones. This extra CPU capacity allows for faster leveled compaction, which is used by SOPHIA's plan (while Static-Opt uses size-tiered compaction), and hence leads to greater performance difference for reads.

We also evaluate SOPHIA with the same workload when the data volume per node increases. We vary the amount of data loaded initially into each node (in a cluster of 4 nodes) and measure the gain over Static-Opt in Figure 8. For the 30GB case, the data volume grows beyond the RAM size of the used instances (M4.xlrage with 16 GB RAM). We notice that the gain from applying SOPHIA's reconfiguration plan is consistent with increasing the data volume from 3 GB to 30 GB. We also notice that the gain increases for the case of 30 GB. This is also due to the different compaction methods used by Static-Opt (size-tiered) and SOPHIA (Leveled compaction), the later can provide better read performance with increasing data volumes. However, this benefit of Leveled compaction was not captured by RAFIKI predictions, which was trained on a single node with 6 GB of data. This can be addressed by either replacing RAFIKI by a data volume-aware static tuner, or re-training RAFIKI when a significant change in data volume per node occurs.

Experiment 5: Varying RF and CL

We evaluate the impact of applying SOPHIA to clusters with different RF and CL values. We use the HPC workload with 5 concurrent jobs. We fix the number of nodes to 8 and vary RF and CL as shown in Figure 7 (CL quorum implies CL = $\lceil RF/2 \rceil$). We notice that SOPHIA continues to achieve better performance than all 3 static baselines for all RF, CL values. For RF=1, CL=1, we use SOPHIA-AGGRESSIVE because when RF=CL, we cannot reconfigure the cluster without degrading availability. The key observation is that SOPHIA's performance gets closer to the *Theoretical best* as RF-CL becomes higher (compare the RF=3,CL=1 to the RF=5,CL=1 case). This is because the number of steps SOPHIA needs to perform the reconfiguration is inversely proportional to RF-CL as discussed in Sec. 4.5). This allows SOPHIA to react faster to changes in the applied workload and thus achieve better performance. Moreover, we notice that the performance of the cluster degrades with increasing RF or CL. Increasing RF increases the data volume stored by each node, which increases the number of SSTables and hence reduces the read performance. Also increase in CL requires more nodes to respond to each request before acknowledgment to the client, which also reduces the performance.

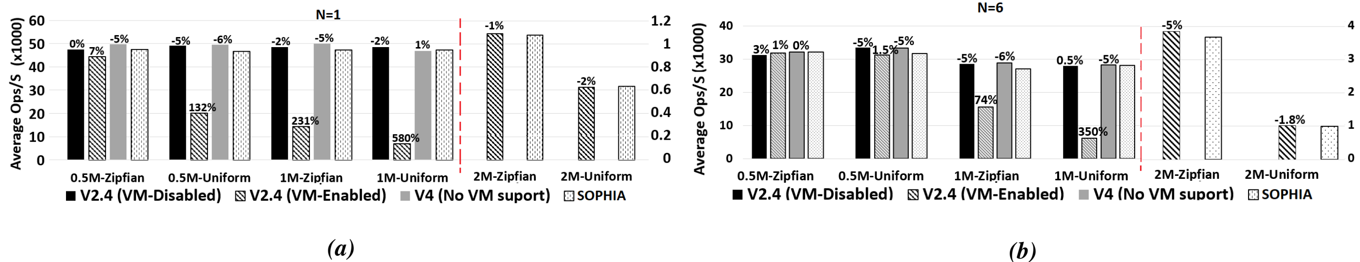
Experiment 6: Noisy Workload Predictions

We show how sensitive SOPHIA is to the level of noise in the predicted workload pattern. We use the HPC workload with 5 concurrent jobs. In HPC queues, there are two typical sources of such noise—an impatient user removing a job from the queue and the arrival of hitherto unseen jobs. We add noise to the predicted workload pattern $\sim U(-R,R)$, where R gives the level of noise. The resulting value is bounded between 0 and 1.

From Figure 9, we see that adding noise to SOPHIA slightly reduces its performance. However, such noise will not cause significant changes to SOPHIA's optimal reconfiguration plan. This is because SOPHIA treats each entry in the reconfiguration plan as a binary decision, i.e., reconfigure if Benefit $\geq$ Cost. So even if the values of both Benefit and Cost terms change, the same plan takes effect as long as the inequality still holds. This allows SOPHIA to achieve significant improvements for long-term predictions even with high noise levels.

Experiment 7: Redis Case Study

We now show a case study with the popular NoSQL database Redis, which has a long-standing pain point in setting a performance-critical parameter against changing workloads. Large-scale processing frameworks such as Spark can de-



(a) *(b)*

Figure 10: Impact of tuning Redis' VM configuration parameters with SOPHIA ***with the data analytics workload. The percentage improvement of*** SOPHIA ***is shown on each bar and the right Y-axis is for the 2M jobs. A missing bar represents a failed job. We notice that the current Redis fails for large workloads (2M), while*** SOPHIA ***achieves the best of both worlds***

liver much higher performance when combined with Redis due to its shared distributed memory infrastructure [67, 38]. Redis is an in-memory data store (stores all keys and values in memory) while writing to persistent storage is only supported for durability. However, in its earlier versions (till V2.4), Redis used to offer a feature called *Virtual Memory* [36]. This feature allowed Redis to work on datasets larger than the available memory by swapping rarely used values to disk, while keeping all keys and hot values in memory. Since V2.4, this feature was removed as it caused serious performance degradation in many Redis deployments due to non-optimal setting as reflected in many posts in discussion forums [59, 25, 60]. We use SOPHIA to tune this feature and compare the performance to three baselines: (1) Redis V2.4 with VM-disabled (Default configuration), (2) Redis V2.4 with VM-enabled, (3) Redis V4 with default configuration (no VM support, most production-proven).

To tune Redis' *VM*, we investigate the impact of two configuration parameters: (1) *vm-enable*: a Boolean parameter that enables or disables the feature. (2) *vm-max-memory*: the memory limit after which Redis starts swapping least-recently-used values to disk. These features cannot be reconfigured without a server restart.

We tune the performance of Redis for simulated data analytics workloads that vary with respect to job sizes and access patterns. We use the popular YCSB (Yahoo! Cloud Serving Benchmark) tool [12] to simulate HPC workloads as in [28, 24]. We collect 128 data points for jobs that vary with respect to their sizes (0.5M, 1M, 2M), their access patterns (i.e., read-heavy vs write-heavy) and also their request distribution (i.e., Uniform vs Zipfian). We use 75% of the data points (selected uniformly) to train a linear regression model and 25% for testing. The model provides accurate prediction of throughput for any job and configuration (avg. R^2=0.92). Therefore, we use this simpler model in place of Rafiki.

Redis can operate in `Stand-alone` mode as well as a `Cluster` mode [37]. In `Cluster` mode, data is automatically sharded across multiple Redis nodes. We show the gain of using our system with Redis for both modes of operation. No replication is used for `Stand-alone` mode. Whereas for `Cluster` mode, we use a replication factor of 1 (i.e., a single slave per master). We use AWS servers of type C3.Large with 2 vCPUs and 3.75GB RAM each. Selecting such a small RAM server demonstrates the advantage of using *VM* with jobs that cannot fit in memory—1.8M records fit in the memory. We evaluate the performance SOPHIA on a single server (Figure 10a) as well as a cluster of 6 servers with 3 masters and 3 slaves (Figure 10b) and report the average throughput per server. From Figure 10 we see that for all record sizes and request distributions, SOPHIA performs the best or close to the best. If records fit in memory, then the no VM configuration is better. For Uniform distribution, VM performs worst, because records often have to be fetched from disk. If records do not fit in memory, the no VM options (including the latest Redis) will simply fail (hence the lack of a bar for 2.0M records). Thus, SOPHIA, by automatically selecting the right parameters for changing workloads, can achieve the best of both worlds: fast in-memory database, and leverage disk in case of spillover.

7 Related Work

Reconfiguration for dynamic workloads. A few systems such as Rafiki [45], Outtertune [64], and SmartConf [65] have been proposed to automatically find the optimal software configurations for a given workload. All these systems assume that the workload change is a long-term, and for which reconfiguring the system is always beneficial. However, we show that in many real-world workloads, both long-term and short-term changes are observed and therefore SOPHIA decides when and how to apply the new configurations to achieve globally optimal performance, while respecting the user's availability requirements.

Reconfiguration in databases. Several works proposed online reconfiguration for databases where the goal is not to update the configuration settings, but to control how the data is distributed among multiple server instances [14, 6, 22, 18, 66]. Among these, Morphus [22] targets MongoDB but cannot handle Cassandra due to its peer-to-peer topology and sharding. Tuba [5] reconfigures geo-replicated key-value stores by changing locations of primary and secondary replicas to improve overall utility of the storage system. Rocksteady [34] is a data migration protocol for in-memory databases to keep tail latency low with respect to workload changes. However, no parameter tuning

or cost-benefit analysis is involved. A large body of work focused on choosing the best logical or physical design for static workloads in DBMS [13, 10, 70, 26, 11, 63, 2, 53, 54]. Another body of work improves performance for static workloads by finding correct settings for DBMS performance knobs [17, 16, 45, 69, 64]. SOPHIA performs *online* reconfiguration of the performance tuning parameters of distributed databases for *dynamic* workloads.

Reconfiguration in distributed systems and clouds. Several works have addressed the problem in the context of traditional distributed systems [29, 3] and cloud platforms [41, 68, 47, 46]. Some solutions present a theoretical approach, reasoning about correctness for example [3], while some present a systems-driven approach such as performance tuning for MapReduce clusters [41, 4]. BerkeleyDB [52] models probabilistic dependencies between configuration parameters. A recent work, *SmartConf* [65] provides a rigorous control-theoretic approach to continuously tune a distributed application in an application-agnostic manner. However, it cannot consider dependencies among the performance-critical parameters and cannot handle categorical parameters.

8 Conclusion

Current static tuners can provide close to optimal configuration for a static workload. However, they cannot determine whether and when to perform a configuration switch to maximize benefit over a future time horizon with changing workloads. We design SOPHIA to perform such reconfiguration while maintaining data availability and respecting the consistency level requirement. Our fundamental technical contribution is a cost-benefit analysis that analyzes the relative cost and the benefit of each reconfiguration action and determines a reconfiguration plan for a future time window. It then develops a distributed protocol to gracefully switch over the cluster from the old to the new configuration. We find benefits of SOPHIA applied to three distinct workloads (a metagenomics portal, a bus-tracking application, and a data analytics workload) over the state-of-the-art static tuners, for two NoSQL databases, Cassandra and Redis. Our work uncovers two big open challenges. How to do anticipatory configuration changes for future workload patterns? How to handle heterogeneity in the cluster, i.e., one where each server instance may have its own configuration and may contribute differently to the overall performance?

Acknowledgement

We thank our shepherd Asaf Cidon and all the reviewers for their insightful comments. This work is supported in part by NSF grant 1527262, NIH Grant 1R01AI123037, Lilly Endowment (Wabash Heartland Innovation Network - WHIN), and Adobe Research. This material was in part based upon research supported by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research, under contract DE-AC02-06CH11357. The funders had no role in the design or execution of the work. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

9 Supplemental Material

9.1 Cost-Benefit Analysis Derivation

Qualitatively, the benefit summed up over the time window is the increase in throughput due to the new optimal configuration option relative to the current configuration option.

$$B = \sum_{k \in [0, T_L]} H_{\text{sys}}(\boldsymbol{W}(k), \boldsymbol{C}^T_{\text{sys}}(k)) \tag{7}$$

where $\boldsymbol{W}(k)$ is the k-th element in the time-varying workload vector $\boldsymbol{W}$ and $\boldsymbol{C}^T_{\text{sys}}$ is the time-varying system configuration derived from $\boldsymbol{C}^{\Delta}_{\text{sys}}$. Likewise, the cost summed up over the time window is the loss in throughput incurred during the transient period of reconfiguration.

$$\begin{aligned} L &= \sum_{k \in [1,M]} \frac{\cancel{R}}{\cancel{N_s}} \cdot H_{\text{sys}}(\boldsymbol{W}(t_k), \boldsymbol{C}_k) \cdot \frac{\cancel{N_s}}{\cancel{R}} \cdot T_r \\ &= \sum_{k \in [1,M]} H_{\text{sys}}(\boldsymbol{W}(t_k), \boldsymbol{C}_k) \cdot T_r \end{aligned} \tag{8}$$

where $\boldsymbol{C}_k$ the configuration specified by the k-th entry of the reconfiguration plan $\boldsymbol{C}^{\Delta}_{\text{sys}}$, and T_r is the number of seconds a single server is offline during reconfiguration.

9.2 Synthetic HPC Workloads

For long-horizon reconfiguration plans, we simulate synthetic workloads representative of batch data analytics jobs, submitted to a shared HPC queue. We integrate SOPHIA with a job scheduler (like PBS [27]), that examines jobs while they wait in a queue prior to execution. Thus, the scheduler can profile the jobs waiting in the queue, and hence forecast the aggregate workload over a lookahead horizon, which is equal to the length of the queue. We model the jobs on data analytics jobs submitted to a Microsoft Cosmos cluster [21]

Figure 11 shows the simulated workload pattern for HPC Analytics case. We vary the level of concurrency and collect the aggregate workload observed by the NoSQL datastore.

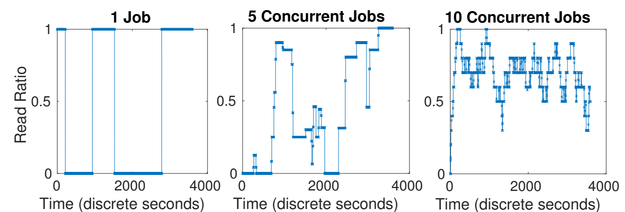


Figure 11: Simulated Workload patterns for 1, 5, and 10 concurrent jobs

References

[1] AGARWAL, S., KANDULA, S., BRUNO, N., WU, M.-C., STOICA, I., AND ZHOU, J. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 21–21.

[2] AGRAWAL, S., NARASAYYA, V., AND YANG, B. Integrating vertical and horizontal partitioning into automated physical database design. In *ACM SIGMOD international conference on Management of data* (2004).

[3] AJMANI, S., LISKOV, B., AND SHRIRA, L. Modular software upgrades for distributed systems. *ECOOP 2006–Object-Oriented Programming* (2006), 452–476.

[4] ANANTHANARAYANAN, G., AGARWAL, S., KANDULA, S., GREENBERG, A., STOICA, I., HARLAN, D., AND HARRIS, E. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems* (2011), ACM, pp. 287–300.

[5] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *OSDI* (2014), pp. 367–381.

[6] BARKER, S. K., CHI, Y., HACIGÜMÜS, H., SHENOY, P. J., AND CECCHET, E. Shuttledb: Database-aware elasticity in the cloud. In *ICAC* (2014), pp. 33–43.

[7] CARLSON, J. L. *Redis in action*. Manning Publications Co., 2013.

[8] CASSANDRA. Cassandra. `http://cassandra.apache.org/`, September 2018.

[9] CHATERJI, S., KOO, J., LI, N., MEYER, F., GRAMA, A., AND BAGCHI, S. Federation in genomics pipelines: techniques and challenges. *Briefings in bioinformatics 20*, 1 (2017), 235–244.

[10] CHAUDHURI, S., AND NARASAYYA, V. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 3–14.

[11] CHAUDHURI, S., AND NARASAYYA, V. R. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB* (1997).

[12] 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* (2010), ACM, pp. 143–154.

[13] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *VLDB Endowment* (2010).

[14] DAS, S., NISHIMURA, S., AGRAWAL, D., AND EL ABBADI, A. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *VLDB Endowment* (2011).

[15] DCSL. Rafiki configuration tuner middleawre. `https://engineering.purdue.edu/dcsl/software/`, February 2018.

[16] DEBNATH, B. K., LILJA, D. J., AND MOKBEL, M. F. Sard: A statistical approach for ranking database tuning parameters. In *IEEE International Conference on Data Engineering Workshop (ICDEW)* (2008).

[17] DUAN, S., THUMMALA, V., AND BABU, S. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment 2*, 1 (2009), 1246–1257.

[18] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *ACM SIGMOD International Conference on Management of data* (2011).

[19] FALOUTSOS, C., GASTHAUS, J., JANUSCHOWSKI, T., AND WANG, Y. Forecasting big time series: old and new. *Proceedings of the VLDB Endowment 11*, 12 (2018), 2102–2105.

[20] FEATHERSTON, D. Cassandra: Principles and application. *Department of Computer Science University of Illinois at Urbana-Champaign* (2010).

[21] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys)* (2012), ACM, pp. 99–112.

[22] GHOSH, M., WANG, W., HOLLA, G., AND GUPTA, I. Morphus: Supporting online reconfigurations in sharded nosql systems. *IEEE Transactions on Emerging Topics in Computing* (2015).

[23] GIFFORD, D. K. Weighted voting for replicated data. In *SOSP* (1979).

[24] GREENBERG, H., BENT, J., AND GRIDER, G. {MDHIM}: A parallel key/value framework for {HPC}. In *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)* (2015).

[25] GROUPS.GOOGLE. Problem with restart redis with vm feature on. https://groups.google.com/forum/#!topic/redis-db/EQA0WdvwghI, March 2011.

[26] GUPTA, H., HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Index selection for olap. In *IEEE International Conference on Data Engineering (ICDE)* (1997).

[27] HENDERSON, R. L. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1995), Springer, pp. 279–294.

[28] JIA, Z., WANG, L., ZHAN, J., ZHANG, L., AND LUO, C. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC)* (2013), IEEE, pp. 66–76.

[29] KEMME, B., BARTOLI, A., AND BABAOGLU, O. Online reconfiguration in replicated databases based on group communication. In *Dependable Systems and Network (DSN)* (2001), IEEE, pp. 117–126.

[30] KHANDELWAL, A., AGARWAL, R., AND STOICA, I. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI* (2016), pp. 485–500.

[31] KIM, S. G., HARWANI, M., GRAMA, A., AND CHATERJI, S. EP-DNN: a deep neural network-based global enhancer prediction algorithm. *Scientific reports 6* (2016), 38433.

[32] KIM, S. G., THEERA-AMPORNPUNT, N., FANG, C.-H., HARWANI, M., GRAMA, A., AND CHATERJI, S. Opening up the blackbox: an interpretable deep neural network-based classifier for cell-type specific enhancer predictions. *BMC systems biology 10*, 2 (2016), 54.

[33] KOO, J., ZHANG, J., AND CHATERJI, S. Tiresias: Context-sensitive approach to decipher the presence and strength of MicroRNA regulatory interactions. *Theranostics 8*, 1 (2018), 277.

[34] KULKARNI, C., KESAVAN, A., ZHANG, T., RICCI, R., AND STUTSMAN, R. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 390–405.

[35] LAB, A. N., AND OF CHICAGO, U. MG-RAST's m5nr-table schema, 2019. [Online; accessed 1 29-August-2018].

[36] LABS, R. Redis Virtual Memory. https://redis.io/topics/virtual-memory, 2018. [Online; accessed 1-September-2018].

[37] LABS, R. Redis Cluster. https://redis.io/topics/cluster-tutorial, 2019. [Online; accessed 1-May-2019].

[38] LABS, R. Spark-Redis: Analytics Made Lightning Fast. https://redislabs.com/solutions/use-cases/spark-and-redis/, 2019. [Online; accessed 1-May-2019].

[39] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[40] LEIPZIG, J. A review of bioinformatic pipeline frameworks. *Briefings in bioinformatics 18*, 3 (2017), 530–536.

[41] LI, M., ZENG, L., MENG, S., TAN, J., ZHANG, L., BUTT, A. R., AND FULLER, N. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 165–176.

[42] MA, L. Tiramisu: Dataset for bus tracking applications. http://www.cs.cmu.edu/~malin199/data/tiramisu-sample/, June 2018. [Online; accessed 1-September-2018].

[43] MA, L., VAN AKEN, D., HEFNY, A., MEZERHANE, G., PAVLO, A., AND GORDON, G. J. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)* (2018), ACM, pp. 631–645.

[44] MAHADIK, K., WRIGHT, C., ZHANG, J., KULKARNI, M., BAGCHI, S., AND CHATERJI, S. SARVAVID: A domain specific language for developing scalable computational genomics applications. In *International Conference on Supercomputing* (2016), ACM, p. 34.

[45] MAHGOUB, A., WOOD, P., GANESH, S., MITRA, S., GERLACH, W., HARRISON, T., MEYER, F., GRAMA, A., BAGCHI, S., AND CHATERJI, S. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th International ACM/IFIP/USENIX Middleware Conference* (2017), pp. 1–13.

[46] MAJI, A. K., MITRA, S., AND BAGCHI, S. Ice: An integrated configuration engine for interference mitigation in cloud services. In *EEE International Conference on Autonomic Computing (ICAC)* (2015).

[47] MAJI, A. K., MITRA, S., ZHOU, B., BAGCHI, S., AND VERMA, A. Mitigating interference in cloud services by middleware reconfiguration. In *ACM International Middleware Conference* (2014).

[48] MEDIUM. Cassandra: Distributed key-value store optimized for write-heavy workloads. https://medium.com/coinmonks/cassandra-distributed-key-value-store-optimized-for-write-heavy-workloads-77f69c01388c. [Online; accessed 1-May-2019].

[49] MEYER, F., BAGCHI, S., CHATERJI, S., GERLACH, W., GRAMA, A., HARRISON, T., PACZIAN, T., TRIMBLE, W. L., AND WILKE, A. MG-RAST version 4lessons learned from a decade of low-budget ultra-high-throughput metagenome analysis. *Briefings in bioinformatics* (2017).

[50] MEYER, F., BAGCHI, S., CHATERJI, S., GERLACH, W., GRAMA, A., HARRISON, T., TRIMBLE, W., AND WILKE, A. Mg-rast version 4–lessons learned from a decade of low-budget ultra-high-throughput metagenome analysis. *Briefings in Bioinformatics 105* (2017).

[51] MOZAFARI, B., CURINO, C., JINDAL, A., AND MADDEN, S. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 acm sigmod international conference on management of data* (2013), ACM, pp. 301–312.

[52] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley db. In *USENIX Annual Technical Conference* (1999), pp. 183–191.

[53] PAVLO, A., JONES, E. P., AND ZDONIK, S. On predictive modeling for optimizing transaction execution in parallel oltp systems. *VLDB Endowment* (2011).

[54] RAO, J., ZHANG, C., MEGIDDO, N., AND LOHMAN, G. Automating physical database design in a parallel database. In *ACM SIGMOD international conference on Management of data* (2002).

[55] SCHEFFE, H. *The analysis of variance*, vol. 72. John Wiley & Sons, 1999.

[56] SCYLLADB. Scylla vs. Cassandra benchmark. http://www.scylladb.com/technology/cassandra-vs-scylla-benchmark-2/, October 2015.

[57] SCYLLADB. ScyllaDB. http://www.scylladb.com/, September 2017.

[58] SEARS, R., AND RAMAKRISHNAN, R. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 217–228.

[59] SERVERFAULT. Should I use vm or set the maxmemory. https://serverfault.com/questions/432810/should-i-use-vm-or-set-the-maxmemory-with-redis-2-4, September 2012.

[60] STACKOVERFLOW. Redis Virtual Memory in 2.6. https://stackoverflow.com/questions/9205597/redis-virtual-memory-in-2-6, February 2012.

[61] SULLIVAN, D. G., SELTZER, M. I., AND PFEFFER, A. *Using probabilistic reasoning to automate software tuning*, vol. 32. ACM, 2004.

[62] TRAN, D. N., HUYNH, P. C., TAY, Y. C., AND TUNG, A. K. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)* (2008).

[63] VALENTIN, G., ZULIANI, M., ZILIO, D. C., LOHMAN, G., AND SKELLEY, A. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *IEEE International Conference on Data Engineering (ICDE)* (2000).

[64] VAN AKEN, D., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1009–1024.

[65] WANG, S., LI, C., HOFFMANN, H., LU, S., SENTOSA, W., AND KISTIJANTORO, A. I. Understanding and auto-adjusting performance-sensitive configurations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2018).

[66] WEI, X., SHEN, S., CHEN, R., AND CHEN, H. Replication-driven live reconfiguration for fast distributed transaction processing. In *USENIX Annual Technical Conference* (2017).

[67] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[68] ZHANG, R., LI, M., AND HILDEBRAND, D. Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker

containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on* (2015), IEEE, pp. 365–368.

[69] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Symposium on Cloud Computing (SoCC)* (2017).

[70] ZILIO, D. C., AND SEVCIK, K. C. *Physical database design decision algorithms and concurrent reorganization for parallel database systems.* PhD Thesis Citeseer, 1999.

libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)

Soyeon Park Sangho Lee[†] Wen Xu Hyungon Moon[*] Taesoo Kim

Georgia Institute of Technology
[†]*Microsoft Research*
[*]*Ulsan National Institute of Science and Technology*

Abstract

Intel Memory Protection Keys (MPK) is a new hardware primitive to support thread-local permission control on groups of pages without requiring modification of page tables. Unfortunately, its current hardware implementation and software support suffer from security, scalability, and semantic problems: (1) vulnerable to protection-key-use-after-free; (2) providing the limited number of protection keys; and (3) incompatible with `mprotect()`'s process-based permission model.

In this paper, we propose `libmpk`, a software abstraction for MPK. It virtualizes the hardware protection keys to eliminate the protection-key-use-after-free problem while providing accesses to an unlimited number of virtualized keys. To support legacy applications, it also provides a lazy inter-thread key synchronization. To enhance the security of MPK itself, `libmpk` restricts unauthorized writes to its metadata. We apply `libmpk` to three real-world applications: OpenSSL, JavaScript JIT compiler, and Memcached for memory protection and isolation. Our evaluation shows that it introduces negligible performance overhead (<1%) compared with the original, unprotected versions and improves performance by 8.1× compared with the secure equivalents using `mprotect()`. The source code of `libmpk` is publicly available and maintained as an open source project.

1 Introduction

Maintaining and enforcing memory access permission is an important duty of OSes and CPUs. Traditionally, they have used page tables to specify whether processes have rights to read from, write to, or execute specific memory pages. OSes can change access permission by updating page-table entries (PTEs) and flushing corresponding translation lookaside buffer (TLB) entries to reload them. In addition to page tables, some CPUs, e.g., ARM [5] and IBM Power [18], allow OSes to maintain the permission of a *page group* together by assigning the same *key* to the correlated pages and controlling the key's permission. To modify the permission of the page groups, OSes should, on behalf of the process, change the permission of the corresponding registers.

Recently, Intel deployed a similar key-based permission control, called Intel Memory Protection Keys (Intel MPK) [20], that allows a *userspace* process to change the permission of the page groups. MPK has three key benefits over page-table-based mechanisms: (1) performance, (2) group-wise control, and (3) per-thread view. First, MPK utilizes a protection key rights register (`PKRU`) to maintain the access rights of individual keys associated with specific pages: read/write, read-only, or no access. Processes only need to execute a non-privileged instruction (`WRPKRU`) to update `PKRU`, which takes less than 20 cycles (§2.3) and requires no TLB flush and context switching. Note that `PKRU` and page-table permissions cannot override each other, so the *effective permission* is the intersection of both.

Second, MPK can change the access rights of up to 16 different *page groups* at once, where each group consists of pages associated with the same key[1]. This group-wise control allows applications to change access rights to page groups according to the types and contexts of data stored in them (e.g., per-session data of a web server).

Third, MPK allows each thread (i.e., each hyperthread) to have a unique `PKRU`, realizing per-thread memory view. Accordingly, even if two threads share the same address space, their access rights to the same page can be different.

Although MPK is a promising primitive in concept, its current hardware implementation as well as standard library and kernel support suffer from three problems: (1) *security*, (2) *scalability*, and (3) subtle *semantic* differences, hindering its broader adoption. First, we found that MPK suffers from the *protection-key-use-after-free* problem. The Linux kernel provides two system calls, `pkey_alloc()` and `pkey_free()`, to allocate and de-allocate protection keys, respectively. During key de-allocation (`pkey_free()`), however, it does not invalidate pages associated with a de-allocated key, resulting in ambiguity when the de-allocated key is re-allocated and assigned to different pages later.

Second, MPK fails in scaling because `PKRU` can manage only up to 16 protection keys because of its hardware limitation. When an application tries to allocate more than 16 protection keys, `pkey_alloc()` simply fails, implying that the application itself should implement its own mechanism to

[1]The default group (0) has a special purpose, so only 15 groups are available for general uses.

multiplex these protection keys.

Third, the semantic of MPK is different from the conventional `mprotect()`, i.e., thread-view versus process-view, which results in potential security and performance problems. For example, the Linux kernel implements an execute-only memory with MPK by disabling read access through `PKRU` and allowing execution through a page table: `mprotect(addr, len, PROT_EXEC)`. Although this feature is invoked via `mprotect()`, it only changes the `PKRU`'s permission of the calling thread, meaning that other threads sharing the same address space can still read the execute-only memory. In other words, it is non-trivial to apply MPK securely and efficiently to legacy applications that rely on a process-level memory permission model.

In this paper, we propose `libmpk`, a secure, scalable, and semantic-compatible software abstraction to fully utilize MPK in a practical manner. In particular, `libmpk` implements (1) *protection key virtualization* to eliminate the protection-key-use-after-free problem and to support the unrestricted number of memory page groups, (2) *lazy inter-thread key synchronization* to selectively ensure per-process semantics with MPK, allowing us to substitute `mprotect()` in an efficient and compatible manner, and (3) *metadata integrity* to ensure the integrity of the mapping information while minimizing the number of system call invocations. `libmpk` consists of a userspace library mainly for efficient permission change and a kernel module mainly for synchronization and metadata integrity.

To show the effectiveness and practicality, we apply `libmpk` to three real-world applications: OpenSSL library, JavaScript just-in-time (JIT) compiler, and Memcached. First, we modify the OpenSSL library to create secure memory pages for storing cryptographic keys to mitigate information leakage. Second, we modify three JavaScript JIT compilers (i.e., SpiderMonkey, ChakraCore, and v8) to protect the code cache from memory corruption by enforcing the W⊕X security policy. Third, we modify Memcached to secure almost all its data, including the slab and hash table, whose size can be several gigabytes. The evaluation results show that `libmpk` and its applications have negligible overhead (<1%). Furthermore, `libmpk` is 1.73–3.78× faster than `mprotect()` when changing the permission of 1–1,000 pages at the view of a process, and, especially, the throughput of Memcached with `libmpk` is 8.1× higher than that of Memcached with `mprotect()`.

We summarize the contributions of this paper as follows:

- **Comprehensive study.** We study the design, functionality, and characteristics of Intel MPK in detail. We identify the critical challenges of utilizing MPK in terms of security, scalability, and semantics.
- **Software abstraction.** We design and implement `libmpk`, a software abstraction to fully utilize MPK. The protection key virtualization, metadata protection, and inter-thread key synchronization of `libmpk` allow applications to effectively overcome the three challenges.
- **Case studies.** We apply `libmpk` to OpenSSL library, JavaScript JIT compiler, and Memcached to show its effectiveness and practicality. `libmpk` secures them with a few modifications and negligible overhead.

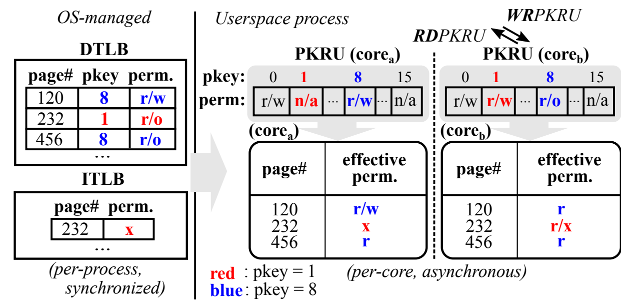


Figure 1: An example showing how MPK checks the permission of a logical core (hyperthread) on a specific memory page according to PKRU and page permissions. The intersection of the permissions determines whether a data access will be allowed. An instruction fetch is independent of the PKRU.

2 Intel MPK Explained

In this section, we describe the hardware design of Intel MPK and current kernel and library support. Also, we check the performance characteristics of MPK to show its efficiency.

2.1 Hardware Primitives

Intel MPK updates the permission of a group of pages by associating a protection key to the group and changing the access rights of the protection key instead of individual memory pages (Figure 1).

Protection key field in page table entry. MPK assigns a unique protection key to a memory page group to update its permission at the same time. MPK exploits the previously unused four bits of each page table entry (from 32nd to 35th bits) to store a memory page's corresponding key value. Thus, MPK supports up to 16 different page groups. Since only supervised code can access and change PTEs, the Linux kernel (from version 4.6) starts to provide a new system call, `pkey_mprotect()`, to allow applications to assign or change the keys of their memory pages (§2.2).

Protection key rights register (`PKRU`). MPK uses the value of `PKRU` to determine its access right to each page group. Two bits representing the right are *access disable* (AD) and *write disable* (WD) bits. The value of (AD, WD) represents a thread's permission to a page group: read/write $(0,0)$, read-only $(0,1)$, or no access $(1,x)$. `PKRU` exists for each hyperthread to provide a per-thread view.

Instruction set. MPK introduces two new instructions to manage the `PKRU`: (1) `WRPKRU` to update the protection information of the `PKRU` and (2) `RDPKRU` to retrieve the current protection information from the `PKRU`. `WRPKRU` uses three registers as input: the `EAX` register containing new protection

Name	Cycles	Description
pkey_alloc()	186.3	Allocate a new pkey
pkey_free()	137.2	Deallocate a pkey
pkey_mprotect()	1,104.9	Associate a pkey key with memory pages
pkey_get()/RDPKRU	0.5	Get the access right of a pkey
pkey_set()/WRPKRU	23.3	Update the access right of a pkey
Ref. mprotect(): 1,094.0 / MOVQ (rbx to rdx): 0.0 / MOVQ (rdx to xmm): 2.09		

Table 1: Overhead of MPK instruction, system calls, and standard library APIs. *ref* shows the overhead of mprotect() and normal register move instructions for comparison. We averaged 10 runs of microbenchmarks, where each one executes individual instruction, system call, or API 10 million times while measuring the latency with the RDTSCP instruction.

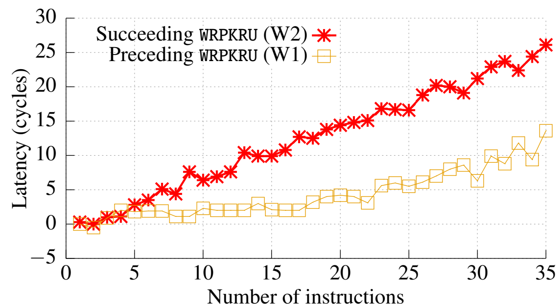


Figure 2: Effect of WRPKRU serialization on simple (i.e., ADD) instructions either preceding or succeeding WRPKRU (average of 10 million repetitions).

information to overwrite the PKRU, and the other two registers, ECX and EDX, filled with zeroes. RDPKRU also uses the three registers for its operation: it returns the current PKRU value via the EAX register while overwriting the EDX register with 0. The ECX register also should be filled with zeroes to execute RDPKRU correctly. Note that the actual usage of ECX and EDX registers is undocumented.

2.2 Kernel Integration and Standard APIs

The Linux kernel has supported MPK since version 4.6, and glibc has supported MPK since version 2.27. They focus on how to manage protection keys and how to assign them to particular PTEs. The Linux kernel provides three new system calls: pkey_mprotect(), pkey_alloc(), and pkey_free(). The kernel also changes the behavior of mprotect() to provide execute-only memory. glibc provides two userspace functions, pkey_get and pkey_set, to retrieve and update the access rights of a given protection key. Table 1 summarizes the APIs.

pkey_mprotect(). The pkey_mprotect() system call extends the mprotect() system call to associate a protection key with the PTEs of a specified memory region while changing its page protection flag. Interestingly, pkey_mprotect() does not allow a user thread to reset a protection key to zero, the default protection key value assigned to newly created memory pages such that it should be public to avoid accidental application crashes. We anticipate that resetting a key to zero is prohibited to avoid such potential crashes made by mistakes (i.e., denying access to the key zero).

pkey_alloc() and pkey_free(). The Linux kernel provides two other new system calls to allocate and de-allocate memory protection keys: pkey_alloc() and pkey_free(). When a user thread invokes pkey_alloc() with access right, the kernel allocates and returns a protection key with corresponding permission according to a 16-bit bitmap that tracks which protection keys are allocated. When a user thread invokes pkey_free(), the kernel simply marks the freed key as available in the bitmap. The pkey_mprotect() function examines the bitmap afterward to prohibit the use of non-allocated keys.

Execute-only memory. The Linux kernel supports execute-only memory with MPK. If a user thread invokes mprotect() only with PROT_EXEC, the kernel (1) allocates a new protection key, (2) disables the read and write permission of the key, and (3) assigns the key to the given memory region.

2.3 Quantifying Characteristics of Intel MPK

To evaluate the overhead and benefits of MPK, we measure (1) the overhead of the MPK instructions, (2) the overhead of the MPK system calls, and (3) the overhead of mprotect() for contiguous memory and sparse memory.

Environment. Our system consists of two Intel Xeon Gold 5115 CPUs (each CPU has 20 logical cores at 2.4 GHz) and 192GB of memory. Linux kernel version 4.14 configured for MPK is installed to this system.

Instruction latency. We measure the latency of RDPKRU and WRPKRU to identify their micro-architectural characteristics. Table 1 summarizes the results. The latency of RDPKRU is similar to that of reading a general register, but the latency of WRPKRU is high. We anticipate that WRPKRU performs serialization (e.g., pipeline flushing) to avoid potential memory access violation resulting from out-of-order execution. To confirm this, we insert various numbers of ADD instructions before (W1) and after (W2) WRPKRU and measure the overall latency (Figure 2). The results show that W2 is always slower than W1, implying that the instructions executed right after WRPKRU fail to benefit from out-of-order execution because of the serialization.

System calls. We measure the latency of the four Linux system calls for MPK (Table 1). The latency of mprotect() and pkey_mprotect() on a 4 KB page is almost the same because they all rely on do_mprotect_pkey() internally. pkey_alloc() and pkey_free() are fast since they involve only simple operations in the kernel, and the domain switching between kernel and userspace dominates their time costs.

Contiguous versus sparse memory pages. Using MPK to change page permission involves only an update on the PKRU and thus is independent of the number of targeted pages and their sparseness. To show the performance benefit of MPK over mprotect(), we check how the number and sparseness of the targeted pages affect the performance of mprotect().

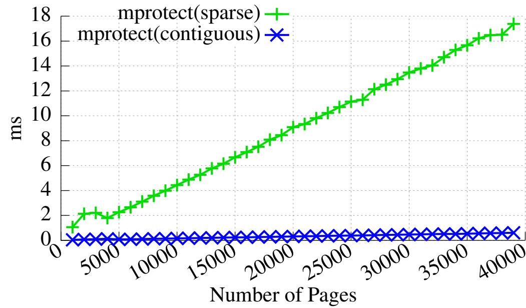


Figure 3: Overhead of `mprotect()` on contiguous and sparse memory (average cost of 10 million repetitions). Protecting contiguous pages takes less time than protecting sparse pages.

To construct contiguous memory pages, we call `mmap()` one time with certain memory size. For sparse memory pages, we call `mmap()` several times with one page size. Figure 3 shows that the overhead of `mprotect()` increases in proportion to the number of pages. The number of pages affects how many virtual memory areas (VMAs) [10] `mprotect()` needs to look up for permission update. Moreover, the overhead of `mprotect()` on sparse memory pages is high because multiple `mprotect()` calls introduce frequent context switchings between kernel and userspace.

Summary. Intel MPK allows a thread to rapidly change the per-thread access rights to a group of pages associated with the same protection key by updating a thread-local register `PKRU` which only takes around 20 cycles. Its performance is independent of the number of pages composing a group and their sparseness, unlike `mprotect()`.

3 Challenges of Utilizing Intel MPK

In this section, we explain the challenges of using MPK in terms of security, scalability, and synchronization.

3.1 Potential Security Problem

The existing OS support of MPK [3] suffers from the protection-key-use-after-free problem. In particular, `pkey_free()` just removes a protection key from a key bitmap and does not update the corresponding PTEs. Regardless of whether a key could already be associated with some pages, the kernel will allocate the key if it is freed by `pkey_free()`. If a program obtains a key that is still associated with some memory pages through `pkey_alloc()`, the new page group will include unintended pages that it is supposed to have. A developer can face this vulnerable situation unconsciously, as current kernel implementation neither handles this automatically nor checks if a free key is still associated with some pages. The developer community also recognized the problem and recommends not to free the protection keys [2, 13]. Handling this problem superficially (i.e., wiping protection keys in PTE) without a fundamental design change of memory management in the kernel will introduce huge performance overhead because it requires traversing the page table and VMAs to detect entries associated with a freed key to update them and flushing all corresponding TLB entries.

3.2 Limited Hardware Resources

Currently, MPK relies on a 32-bit `PKRU` such that it supports up to 16 keys. Developers are responsible for ensuring that an application never creates more than 16 page groups at the same time. This implies that developers have to examine at runtime the number of active page groups, which are used by both the application itself and the third-party libraries it depends on. Otherwise, the program may fail to properly benefit from MPK. This issue undermines the usability of MPK and discourages developers from utilizing it actively. Using a large register (e.g., 1024 bits) does not scale because MPK needs additional storage to associate keys with pages. For example, to support 512 protection keys, nine bits are necessary for each PTE, requiring enlarged page tables, shrunken address bits, or separate storage.

3.3 Semantic Differences

To change the permission of any page group, MPK modifies the value of the `PKRU`. However, the value is effective only in a single thread because `PKRU` is thread-local intrinsically as a register. As a result, different threads in a process can have different permissions for the same page group. This thread-local inherence helps to improve security for the applications that require isolation on memory access among different threads, but hinders MPK from optimizing and improving `mprotect()`. `mprotect()` semantically guarantees that page permissions are synchronized among all threads in a process on which particular applications rely. This not only makes it difficult to accelerate `mprotect()` with MPK, but also breaks the guarantee of execute-only memory implemented on mprotect in the latest kernel. `mprotect()` supporting executable-only memory relying on MPK does not consider synchronization among threads, which developers basically expect of `mprotect()`. Even when the kernel successfully allocates a key for the execute-only page, another thread might have a read access to it due to a lack of synchronization. To make MPK a drop-in replacement of `mprotect()` for both security and usability, developers need to synchronize the `PKRU` values among all the threads.

4 Software Abstraction of libmpk

`libmpk` provides a secure and usable abstraction for MPK by overcoming the challenges (§3). A developer can use MPK easily by either adding calls to `libmpk` APIs or replacing existing `mprotect()` calls with those of `libmpk`. By decoupling the protection keys from APIs, `libmpk` is immunized against protection-key-use-after-free. Also, `libmpk` allows an application to create more than 16 page groups by virtualizing the protection keys and provides a lightweight inter-thread `PKRU` synchronization mechanism. Figure 4 illustrates an overview of `libmpk`. The current version of `libmpk` consists of 1.5k

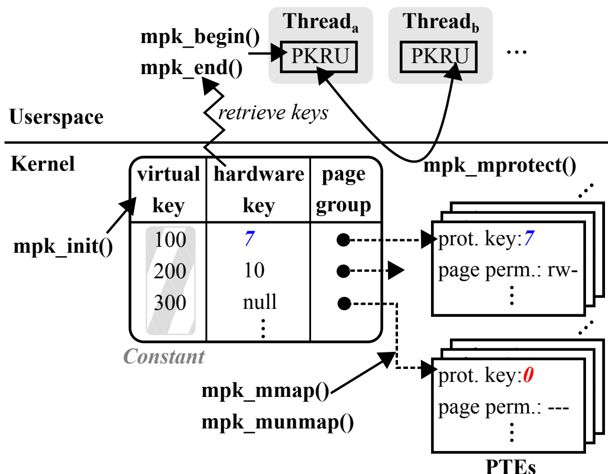


Figure 4: `libmpk` overview. `mpk_init()` pre-allocates hardware keys and initializes the metadata table. `mpk_mmap()` creates a page group with metadata, and `mpk_munmap()` destroys the page group and the corresponding metadata. `mpk_begin()` and `mpk_end()` provide domain-based thread-local isolation. `mpk_mprotect()` synchronizes permission changes globally.

```
1  #define GROUP_1 100
2  #define GROUP_2 101
3
4  int domain_based_isolation () {
5    mpk_init(-1); // default eviction rate: 100%
6    char* addr = (char *)mpk_mmap(GROUP_1, NULL, 0x1000,
7      PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
8    // page permission: rw- & pkey permission: --
9
10   mpk_begin(GROUP_1, PROT_READ | PROT_WRITE);
11   // page permission: rw- & pkey permission: rw
12
13   // write data in GROUP_1
14
15   mpk_end(GROUP_1);
16   // page permission: rw- & pkey permission: --
17
18   printf("%s\n", addr); // SEGMENTATION FAULT
19 }
20
21 int quick_permission_change () {
22   mpk_init(0.5); // set cache eviction rate: 50%
23   void* addr = mpk_mmap(GROUP_2, NULL, 0x1000,
24     PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
25   // page permission: rw- & pkey permission: --
26
27   mpk_mprotect(GROUP_2, PROT_READ | PROT_WRITE | PROT_EXEC);
28   // page permission: rwx & pkey permission: rw
29 }
```

Figure 5: Example code for `libmpk` APIs.

Name	Argument	Description
`mpk_init()`	`evict_rate`	Initialize `libmpk` with an eviction rate
`mpk_mmap()`	`vkey`, `addr`, `len`, `prot` `flags`, `fd`, `offset`	Allocate a page group for a virtual key
`mpk_munmap()`	`vkey`	Unmap all pages related to a given virtual key
`mpk_begin()`	`vkey`, `prot`	Obtain thread-local permission for a page group
`mpk_end()`	`vkey`	Release the permission for a page group
`mpk_mprotect()`	`vkey`, `prot`	Change the permission for a page group globally
`mpk_malloc()`	`vkey`, `size`	Allocate a memory chunk from a page group
`mpk_free()`	`size`	Free a memory chunk allocated by `mpk_malloc()`

Table 2: `libmpk` APIs.

lines of C/C++ code in total.

Goals. To utilize MPK for domain-based isolation and as a substitute for `mprotect()`, we have to overcome the three challenges: (1) insecure key management, (2) hardware resource limitations, and (3) different semantics from `mprotect()`. `libmpk` adopts two approaches: (1) *key virtualization*, and (2) *inter-thread key synchronization*, which effectively solve the challenges. `libmpk` also protects its internal metadata from corruption.

4.1 Threat Model and Assumptions

`libmpk` has the following threat model and assumptions, in accordance with prior studies [21, 33, 35].

`libmpk` aims to prevent an adversary from reading from or writing in sensitive pages through memory corruption vulnerabilities. `libmpk` achieves this goal by protecting the sensitive pages with new MPK APIs and preventing the adversary from arbitrarily executing the `WRPKRU` instruction. We assume that a program should solely use the `libmpk` APIs to utilize MPK in a controlled manner. That is, the program should not use conventional MPK APIs together with the `libmpk` APIs. Also, any uncontrolled execution of the `WRPKRU` instruction should be prohibited by using existing countermeasures, such as data execution prevention [1], control-flow integrity [4, 22, 23, 38], and call gate [35].

4.2 libmpk API

`libmpk` provides eight APIs, shown in Table 2. To utilize `libmpk`, an application first calls `mpk_init()` to obtain all the hardware protection keys from the kernel and initialize its metadata. `mpk_mmap()` allocates a page group for a virtual key, which should be a constant integer that the developer passes. `mpk_munmap()` destructs a page group by freeing a virtual key for the group and unmaps all the pages. `libmpk` maintains the mappings between virtual keys and pages to avoid scanning all pages at this destruction step. On top of these, `libmpk` also provides simple heap over each page group (`mpk_malloc()` and `mpk_free()`), so that a developer can also use one or more page groups to create a heap memory region for sensitive data.

`libmpk` provides two usage models for developers. The first model, a thread-local domain-based isolation model, allows an application to temporarily grant permission to a page group only for the calling thread. `mpk_begin()` and `mpk_end()` are the APIs for this model, which make a page group accessible and inaccessible, respectively. The second model allows an application to quickly change the access rights to a page group by replacing `mprotect()` with `mpk_mprotect()`. Figure 5 shows an example of utilizing `libmpk` APIs.

4.3 Protection Key Virtualization

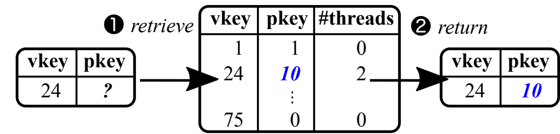
`libmpk` enables an application to create more than 16 page groups by virtualizing the hardware protection keys. When an application creates a new page group by calling `mpk_mmap()`, a virtual key passed as argument is associated with newly

allocated metadata for the new group. The application uses the virtual key to obtain or release the permission, or free the group, while being prohibited from manipulating hardware keys. The exact physical key that a page is associated with is hidden from the program and developer.

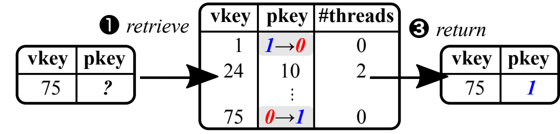
`libmpk` maintains the mappings between virtual and hardware keys through a cache-like structure (Figure 6). If a virtual key is already associated with a hardware key, the virtual key exists inside the cache and further access to it produces a few latencies. Otherwise, `libmpk` evicts another virtual key or does nothing but just invokes `mprotect()` for performance to change page permission. The frequency of eviction or calling `mprotect()` is determined by the eviction rate. The cache guarantees that a frequently updated virtual key will be mapped with a hardware key since it has a high possibility to be placed into the cache.

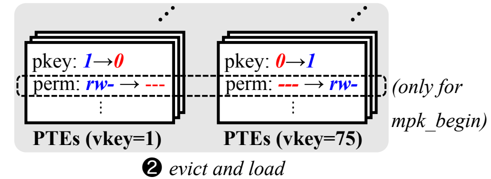
`libmpk` provides two policies to determine the mappings between virtual and hardware keys. When an application grants permission to a page group thread-locally by calling `mpk_begin()`, `libmpk` always maps the group's virtual key with a hardware key and uses it to grant access to the *calling thread*. `libmpk` maintains the mapping until the thread calls `mpk_end()` to release the access. For this reason, `libmpk` does not ensure that a calling thread always obtains the access due to hardware limitations. That is, if all hardware keys are actively used, `libmpk` is no longer able to provide any key. In this case, `mpk_begin()` raises an exception and lets the calling thread handle it (e.g., sleeps until a key is available). If a page group is not used by a thread, `libmpk` evicts the group by changing its protection key to 0 (default) and revoking its page permission to disallow subsequent accesses.

The second policy, `mpk_mprotect()`, also needs to map the virtual key to a hardware key, but not exclusively. Even when the page group is accessible, `libmpk` can unmap a hardware key and rely solely on the page attributes because all threads have the access. Hence, `libmpk` maps only the page groups whose access rights change frequently. If `libmpk` fails to find an available hardware key when it handles `mpk_mprotect()`, it unmaps and uses the least recently used (LRU) key for handling `mpk_mprotect()`. The hardware key of the evicted page group turns to 0. To avoid excessive overhead resulting from frequent unmapping, a developer can configure an *eviction rate* to control whether a hardware key has to be evicted according to how frequently its permission updates. In our approach, enforcing executable-only permission is not straightforward because a conventional approach (i.e., `mprotect()`) does not support executable-only permission. Therefore, `mpk_mprotect()` reserves one key for execute-only pages when an application creates them first, and does not evict this key until all executed-only pages disappear. Every incoming executable-only permission request is guaranteed to get a hardware protection key to achieve executable-only permission. If `mpk_mprotect()` has already been invoked for executable-only page groups, further requests will merge the



(a) Hit case: ❶ A thread calls `mpk_begin()` or `mpk_mprotect()` with a vkey; ❷ `libmpk` returns the corresponding pkey immediately.





(b) Miss case: ❶ A thread retrieves a vkey, but no corresponding pkey exists (pkey=0); ❷ `libmpk` evicts the LRU pkey. In addition, `mpk_begin()` updates the page permission of the evicted and loaded page groups using `mprotect()`; ❸ `libmpk` returns the new pkey.

Figure 6: Key virtualization in `libmpk`. *vkey* and *pkey* represent a virtual key and its corresponding hardware protection key associated with a page group. #threads indicates the number of threads running parallel inside a particular domain.

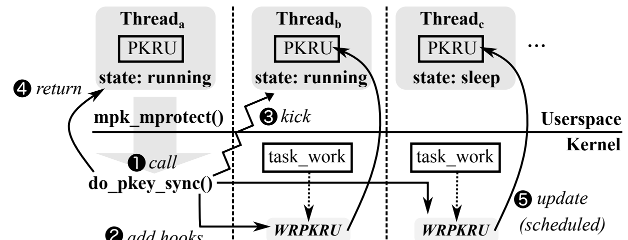


Figure 7: `PKRU` synchronization: ❶ `mpk_mprotect()` calls `do_pkey_sync()` to update the `PKRU` values of remote threads; ❷ `do_pkey_sync()` adds hooks to the threads' `task_work`; ❸ `do_pkey_sync()` kicks all the running threads for synchronization; ❹ `do_pkey_sync()` returns to its caller; ❺ The threads update their `PKRU`s when they are scheduled to run.

incoming page groups with the existing executable-only ones to utilize the reserved key.

The integrity of `libmpk` metadata (e.g., the mappings between virtual and hardware keys, and the page group information) is important to prevent attackers from manipulating `libmpk`'s protection. For the metadata integrity, `libmpk` maps each metadata physical page into two virtual pages: a read-only page for its userspace code and a writable page for its kernel-space code. Updating the metadata can be done only by the `libmpk` kernel module and slightly modified system calls (e.g., `mmap()`, `munmap()`, and `mprotect()`). Most of simple metadata retrieval can be done by the userspace code to avoid unnecessary user-kernel mode switches.

Application	Protection	Protected data	#pkeys	#vkeys	Changed LoC
OpenSSL	Isolation	Private key	1	1	83
JIT (key/page)	W⊕X	Code cache	15	> 15	CC 10 | SM 18
JIT (key/process)	W⊕X	Code cache	1	1	CC 18 | SM 24 | v8 134
Memcached	Isolation	Slab, hashtable	2	2	117

Table 3: Three real-world applications of `libmpk`. To enable W⊕X in JavaScript engines, we use two approaches, including using a virtual key for every page in the code cache (*One key per page*) and using a single protection key for all the pages in the code cache (*One key per process*). **CC**, **SM**, and **v8** indicate Microsoft ChakraCore, Mozilla SpiderMonkey, and Google v8, respectively. **pkeys** and **vkeys** stand for protection keys and virtual keys, respectively.

4.4 Inter-thread Key Synchronization

`libmpk` implements an inter-thread `PKRU` synchronization technique, `do_pkey_sync()`, in `mpk_mprotect()` for two purposes: (1) to ensure no thread has the read access to an execute-only page and (2) to replace existing page-table-based `mprotect()` for performance. `do_pkey_sync()` guarantees that a `PKRU` update is globally visible and effective as soon as it returns. Intuitively, this requires a synchronous inter-thread communication; the calling thread needs to send messages to the other threads and wait until they update the `PKRU` value and acknowledge it, which suffers from a high cost.

We minimize the inter-thread `PKRU` synchronization latency in a *lazy* manner, leveraging the fact that the `PKRU` values are utilized in the userspace. If a remote thread is not currently being scheduled, it does not need the up-to-date `PKRU` value immediately. Even if the thread is currently being scheduled, we only need to update its `PKRU` value when it returns to the userspace. If the calling thread can create a hook that the other threads will invoke right before jumping back to the userspace and ensure that they are not in the userspace, we can guarantee that all the other threads have the new `PKRU` value when `do_pkey_sync()` returns. Figure 7 illustrates the overall procedure of `mpk_mprotect()`. `do_pkey_sync()` utilizes an existing hooking point in the Linux kernel to enforce the remote threads to update the `PKRU` values right before returning to the userspace and ensures that all threads use the new `PKRU` value by sending rescheduling interrupts. In Linux, a thread can register a list of callback functions (`task_work`) that are invoked at designated points (e.g., when returning to the userspace) by calling `task_work_add()`. In this way, `do_pkey_sync()` guarantees that all the remote threads eventually acquire the new `PKRU` value. Although `do_pkey_sync()` still needs to send inter-processor interrupts to ensure that no other thread uses the old `PKRU` value after a certain point, our evaluation shows that the overall latency of `mpk_mprotect()` is less than that of `mprotect()` (§6.2).

5 Applications

We demonstrate the security benefit, efficiency, and usability of `libmpk` by augmenting three types of popular applications: an SSL library, three JavaScript Just-in-time (JIT) compilation engines, and an in-memory key-value store. Table 3 summarizes the mechanisms (e.g., page isolation or W⊕X) that we aim to provide as well as the protected data (e.g., key or code). Evaluation results are described in §6.

5.1 OpenSSL

OpenSSL is a popular open-source library implementing the secure sockets layer (SSL) and transport layer security (TLS) protocols. Since it manages sensitive information (e.g., private keys and encrypted data), its information leakage bugs are security-critical. For example, OpenSSL's Heartbleed bug [26] allowed attackers the chance to leak sensitive data from millions of web servers.

We apply `libmpk` to OpenSSL to protect its private keys from potential information leakage by storing the keys in isolated memory pages. More specifically, the isolated memory pages are protected by single pkey or multiple pkeys assigned per private key to show the trade-off between performance and security. First, we identify all the data types that store private keys (e.g., `EVP_PKEY`) and replace their heap memory allocation function from `OpenSSL_malloc()` to `mpk_malloc()` for single pkey or `mpk_mmap()` for multiple pkeys to store them in an isolated memory region. Next, we locate all the functions that access private keys (e.g., `pkey_rsa_decrypt()`) and modify them to access the isolated memory region by inserting `mpk_begin()` and `mpk_end()` before and after their call sites. Note that assigning pkey per private key offers finer-grained security, which minimizes the attack window for the isolated memory region. For example, even if a function whose call site is located between `mpk_begin()` and `mpk_end()` has a memory leakage bug, it cannot access any other isolated pages except the single page isolated with the pkey provided to `mpk_begin()` as argument.

5.2 Just-in-time (JIT) Compilation

JIT compilation dynamically translates interpreted script languages, e.g., JavaScript and ActionScript, into native machine code or bytecode to avoid the overhead of full compilation and repeated interpretation. Technically, it relies on writable code, resulting in potential arbitrary code execution. To support JIT compilation, the *code cache* that stores code generated at runtime needs to be writable for a JIT compilation thread and be executable for an execution thread. Thus, if attackers compromise the JIT compilation thread, they can make the execution thread execute the code they provide.

ChakraCore [27] and SpiderMonkey [28] mitigate the above-mentioned problem by enforcing the W⊕X security policy on the code cache with `mprotect()`. They make the code cache writable while disallowing execution when they are updating code, and, after it has updated, they make the code cache executable while disallowing write. However, they can suffer from *race condition attacks* [33] because they use `mprotect()` to change page permissions; that is, when a

thread makes the code cache writable with `mprotect()`, other threads compromised by attackers can also manipulate the code cache with the same permission.

We apply `libmpk` to the three popular JavaScript engines (SpiderMonkey, ChakraCore, and v8) to enforce the W⊕X security policy without the race condition problem while ensuring better performance. We propose two approaches to implement the W⊕X policy with `libmpk`.

One key per page. A context-free solution is to replace `mprotect()` with `libmpk` APIs to perform fast permission switches on targeted pages in the code cache. All the protection keys are initialized with read-only permission when a new thread is created. We dedicate *one protection key* to *one page* when it is the first time to be re-protected via `mprotect()` and change its page permission to `rwx`. Later, we only need to call `mpk_begin()` and `mpk_end()` before and after when the JIT compiler updates the corresponding page. Based on the observation that generally only one page is updated at a time, we still invoke `mprotect()` if multiple pages change permission.

One key per process. Another approach is to use a single protection key for the entire code cache. When pages are first committed from the preserved memory region into the code cache, they are assigned with the protection key and their page permission is set to `rwx`. Whenever any page in the code cache is to be updated, the script engine needs to call `mpk_begin()` and `mpk_end()`. Although more pages become temporarily writable, the security of the code cache is ensured thanks to the per-thread view of the protection key.

5.3 In-Memory Key-Value Store

In-memory key-value stores, such as Memcached, are widely used to manage a large amount of data in memory to ensure low latency and high throughput. With a high requirement for performance, such key-value stores normally avoid using security techniques whose performance depends on input size (e.g., `mprotect()` and encryption) to protect stored data. This implies that, if an in-memory key-value store has arbitrary read or write vulnerabilities, attackers are able to leak or corrupt sensitive information stored inside.

`libmpk` manages to efficiently mitigate such attacks. To demonstrate this, we apply `libmpk` to Memcached. `libmpk` protects Memcached's slabs that contain values and hash tables that maintain key-value mappings by replacing Memcached's `malloc()` function with `mpk_malloc()`, and wraps the call sites of all the legitimate functions (e.g., `ITEM_key()` and `assoc_find()`), which operate on protected data with `mpk_begin()` and `mpk_end()`. Note that we assign two different keys to slabs and hash tables, to narrow the attack surface. It is possible to use more keys to secure slabs in a fine-grained manner, e.g., differentiating them according to their sizes. More importantly, `libmpk`'s performance is independent of the size of memory to protect, and thereby efficiently works with Memcached even when protecting data of several gigabytes.

6 Evaluation

In this section, we evaluate `libmpk` in terms of its security implication and performance by answering the following questions:

- What security guarantees does `libmpk` provide? (§6.1)
- Does `libmpk` solve the security, scalability, and semantic-gap problems of existing MPK APIs without introducing much performance overhead? (§6.2)
- Does `libmpk` have negligible performance impact and outperform `mprotect()` in real-world applications? (§6.3)

The same system environment explained in §2.3 is used for performance evaluations.

6.1 Security Evaluation

We first evaluate the security benefits from `libmpk` regarding memory protection and isolation. For OpenSSL and Memcached, `libmpk` provides domain-based isolation to protect memory space that stores sensitive data. The permission for the particular memory space set by `libmpk` is locally effective, which also prevents malicious accesses from other compromised threads. In particular, exploiting a memory corruption bug to leak or ruin sensitive data stored in the isolated pages is killed by segmentation faults resulting from the lack of permission. To verify this, we mimic the Heartbleed vulnerability by deliberately introducing a heap-out-of-bounds read bug and inserting a decoy private key placed next to the victim heap region. When the vulnerability is triggered, OpenSSL hardened by `libmpk` crashes with invalid memory access. However, `libmpk` cannot fully mitigate memory leakage that originates inside the protected domain. Thus, developers should carefully design the domain to minimize the potential attack surface when using `libmpk` in their applications.

JavaScript JIT compilers can use `libmpk` to guarantee W⊕X for JIT code pages. Unlike `mprotect()`, `libmpk` is immune to race condition attacks launched by compromised threads running in parallel resulting from the thread-local effectiveness of protection keys. When the JIT compiler uses `libmpk` to switch the permission of a code page for updates, other threads controlled by attackers cannot write malicious shellcode into the page simultaneously. To verify this, we introduce two custom JavaScript APIs for arbitrary memory read and write to SpiderMonkey and ChakraCore, and test a simple PoC that leverages these two APIs to locate a JIT code page and write shellcode into it. Both engines crash with a segmentation fault at the end.

6.2 Microbenchmarks

We run several microbenchmarks to understand the performance behavior of APIs in `libmpk`.

Cache performance. `libmpk` introduces a cache to enable protection on more than 16 page groups, whose performance

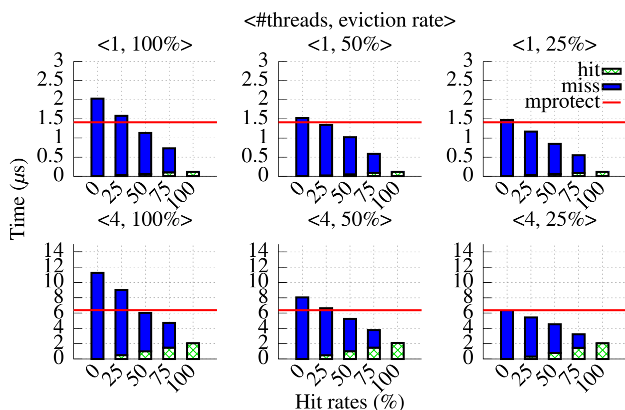


Figure 8: Latency of `libmpk`'s key cache with various hit rates, eviction rates, and different number of threads. `mpk_mprotect()` and `mprotect()` are invoked on a 4 KB page. Red line marks the overhead of `mprotect()`. When the hit rate is 100%, `mpk_mprotect()` is 12.2× faster than `mprotect()` for one thread and 3.11× faster for four threads.

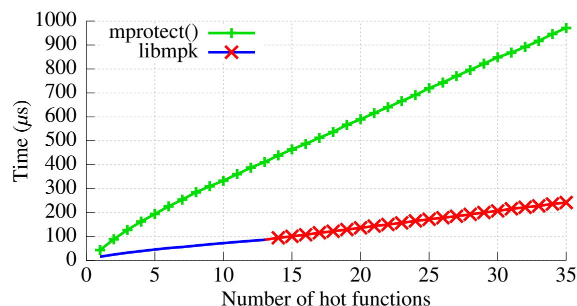


Figure 9: Average time cost to update permission when original and modified ChakraCore JIT-compile an increasing number of hot functions demanding distinct virtual keys.

is affected by its eviction rate and hit rate and the number of virtual keys in use. We run the following two microbenchmarks to check the cache performance.

Hit rate and eviction rate. The first benchmark measures cache performance with different hit rates, eviction rates, and number of threads. We run the benchmark with both one thread and four threads, where each thread warms up by filling the key cache to evade cold miss and invokes `mpk_mprotect()` on one page for a hundred times after 15 entries are filled. Figure 8 presents the evaluation results, where (1) the green box indicates the overhead incurred by the cache hit, which is dominated by the time cost on `WRPKRU` and maintaining internal data structures; (2) the blue box indicates the overhead incurred by the cache miss, which is dominated by the time cost on key eviction. More specifically, `mpk_mprotect()` needs to unset the protection key that is to be evicted and bind a new virtual key to it. We test the microbenchmark with three eviction rates that indicate the ratio of cache misses that eventually leads to key eviction. If a cache miss occurs without key eviction, `mprotect()` is invoked to change the permission of the pages.

Experimental results show that `mpk_mprotect()` outperforms `mprotect()` except when the cache hit rate is below 25% with an eviction rate above 50%. This is because, unlike `mprotect()`, `mpk_mprotect()` does not merge and split the VMAs of targeted pages. It becomes slow when being tested with four threads, but is still comparable with `mprotect()`, whose latency also increases in a multi-threading program.

Number of virtual keys. To evaluate how the number of used virtual keys affects the cache performance of `libmpk`, we re-implement W⊕X in ChakraCore in a *one-key-per-page* approach (see §5.2) and set the eviction rate as 100%. To introduce an increasing number of pages to be protected (i.e., an increasing number of virtual keys to be used) during the execution of ChakraCore, we design a simple microbenchmark. The microbenchmark consists of a set of JavaScript files, and the *i*th file contains *i* hot functions being invoked for 100,000 times. For each hot function, ChakraCore allocates one new executable page to store the native code and performs nine permission switches on the page through one virtual key at runtime. Without any hot function, ChakraCore allocates one page in the code cache. We run the original ChakraCore (version 1.9.0.0-beta) and the modified one with our microbenchmarks, and record the time cost of changing permission of the pages in the code cache (i.e., the execution time of `VirtualProtect()` and that of `mpk_begin()` and `mpk_end()`) in total. Each JavaScript file is executed 200 times, and the average time is presented in Figure 9.

The result shows that with the `libmpk`-based implementation of W⊕X, the time cost on permission switches linearly increases when more hot functions are emitted and thus more virtual keys are allocated to protect the code pages of the hot functions. In particular, after 15 virtual keys are allocated (marked in red), the time cost increases slightly faster than before (marked in blue) as a result of cache eviction. Nevertheless, the ChakraCore hardened by `libmpk` still outperforms by 3.2× the original ChakraCore using `mprotect()` to enforce W⊕X.

Memory overhead. `libmpk` dedicates memory space to store its internal data structures for maintaining the metadata of these page groups under protection (see §4.3). Each `mpk_mmap()` allocates 32 bytes of memory to store the information of a new page group (e.g., base address and size). `libmpk` maintains a hashmap to store the mapping between virtual keys and hardware keys for fast query and access. In the current implementation, we pre-allocate 32 KB of memory for the hashmap, and its size will automatically expand when a program invokes `mpk_mmap()` more than about 4,000 times.

Synchronization latency. Figure 10 shows the latency of inter-thread permission synchronization using `mpk_mprotect()` and `mprotect()` on memory of varying sizes.

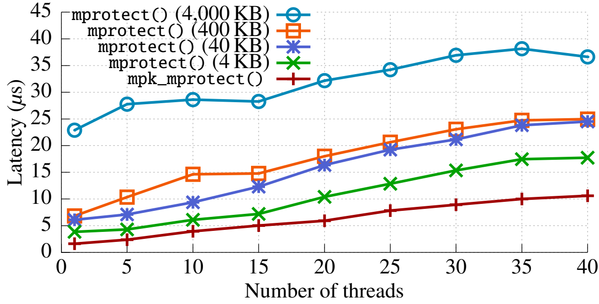


Figure 10: Latency of inter-thread permission synchronization using `mpk_mprotect()` and `mprotect()` calls on memory of varying sizes. `mpk_mprotect()` outperforms `mprotect()` 1.73× for a single page and 3.77× for 1,000 pages.

`mpk_mprotect()` is 1.73× faster than `mprotect()` when updating the permission of a single page. The latency of `mprotect()` increases with the number of pages it changes due to the expensive operations of managing VMAs. Compared to `mpk_mprotect()`, `mprotect()` costs at least 3.78× to change the permission of 1,000 pages. The performance overhead of `mpk_mprotect()` is independent of the number of pages whose permission has been updated. Figure 10 also shows that when there are many threads, the latency of both `mprotect()` and `mpk_mprotect()` increases; `mprotect()` flushes more TLBs, whereas `mpk_mprotect()` creates many hooks in the kernel.

6.3 Application Benchmarks

We measure the performance overhead of `libmpk` in practice by evaluating three applications proposed in §5.

OpenSSL. The Apache HTTP server [12] (`httpd`) uses OpenSSL to implement SSL/TLS protocols. To evaluate the overhead caused by `libmpk`, which is introduced to protect private keys, we use ApacheBench to test `httpd` with both the original OpenSSL library and the modified one with `libmpk`. ApacheBench is launched 10 times and each time sends 1,000 requests of different sizes from four concurrent clients to the server. We choose the DHE-RSA-AES256-GCM-SHA256 algorithm with 1024-bit keys as a cipher suite in the evaluation.

Figure 11 presents the evaluation result. On average, `libmpk` introduces 0.58% and 4.82% performance overhead, respectively, in terms of the throughput. In the single pkey case, the negligible overhead mainly comes from internal data structure maintenance in `libmpk`. In the multiple pkeys case, `httpd` utilizes more than 1,000 pkeys, as it allocates a new pkey while creating a new session. These pkeys are maintained by cache invoke eviction, so the multiple pkeys generates higher overhead than the single pkey case.

Just-in-time compilation. We applied two proposed W⊕X solutions based on `libmpk`, namely, *one key per page* and *one key per process* (§5.2) to both SpiderMonkey (version 59.0) and ChakraCore (version 1.9.0.0-beta) and evaluated their performance with the Octane benchmark [15], which involves heavy JIT-compilation workloads at runtime. Each

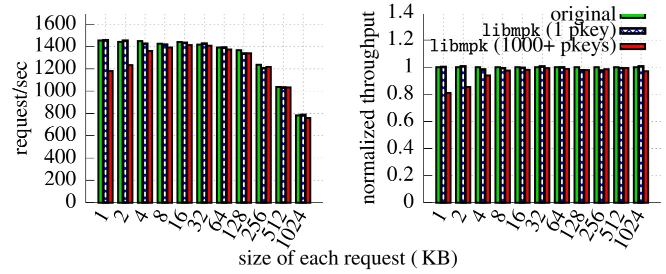


Figure 11: Throughput of original `httpd` and `httpd` hardened by `libmpk`. Protecting private keys with single pkey and 1000+ pkeys, `libmpk` slows down `httpd` by at most 2.52% and 18.84% respectively. The averages of overhead are 0.58% and 4.82% respectively.

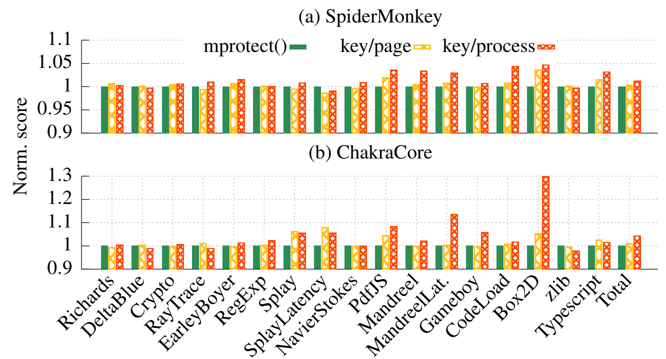


Figure 12: Octane benchmark scores of SpiderMonkey and ChakraCore with original and `libmpk`-based W⊕X solutions. `libmpk` outperforms the original, `mprotect()`-based defense by at most 4.75% (SpiderMonkey) and 31.11% (ChakraCore).

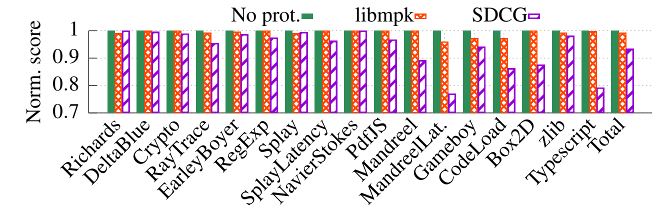


Figure 13: Octane benchmark scores of original v8 and two modified versions of v8 ensuring W⊕X by SDCG and `libmpk`. `libmpk` only introduces 0.81% overall performance overhead for W⊕X in v8, compared with 6.68% caused by SDCG.

JavaScript program in the benchmark was directly executed by the original and modified script engines for 20 times, and we calculated the average score (Figure 12).

For SpiderMonkey, both `libmpk`-based approaches outperform the `mprotect()`-based approach on the total score, namely, 0.38% and 1.26%, which is consistent with the claim from Firefox developers that enabling W⊕X with `mprotect()` in SpiderMonkey introduces less than 1% overhead for the Octane benchmark. The reason is that SpiderMonkey is designed to get rid of unnecessary `mprotect()` calls when its JIT compiler works. The performance scores of nearly all

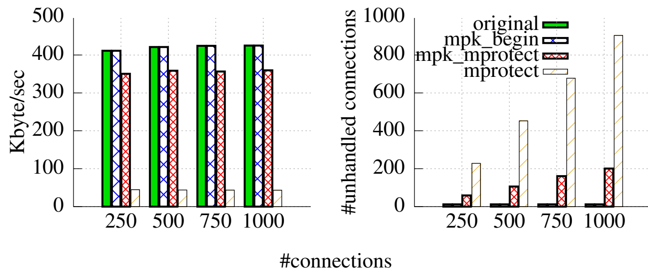


Figure 14: Throughput and unhandled concurrent connections of original Memcached and three versions of Memcached whose key-value pairs are protected by `mpk_begin()`, `mpk_mprotect()`, and `mprotect()`. `mpk_begin()`'s overhead is negligible compared to the original. `mpk_mprotect()` outperforms `mprotect()` 8.1× while ensuring the same semantics.

the programs increase through *one key per page* (at most 3.60% on `Box2D`) and *one key per process* (at most 4.75% on `Box2D`), except for `SplayLatency` protected by *one key per page* (dropped by 1.36%). `SplayLatency` becomes worse because it barely updates the code cache such that the initial overhead to associated keys with pages cannot be hidden.

Our two `libmpk`-based approaches improve ChakraCore by 1.01% and 4.39% on the total score of the Octane benchmark, respectively. ChakraCore is suitable for `libmpk`-based W⊕X solutions since it only makes one page writable per time regardless of emitted code size. *One key per page* increases the performance score of ChakraCore at most 7.96% when testing `SplayLatency`, while *one key per process* improves the performance by at most 31.11% on `Box2D`. Similar to the results of SpiderMonkey, we observe a few performance degradations when benchmarks rarely update code cache.

For v8, we compare our approach with a `mprotect()`-based scheme, SDCG [33]. SDCG protects the JIT code pages of v8 with W⊕X by emitting the code in a dedicated process. No other processes can change the code pages. To demonstrate the performance advantage of our in-process `libmpk`-based approaches, which are free of race condition attacks, we applied one of our approaches, *one key per process*, to Google v8 (version 3.20.17.1 used in [33]) and evaluated the performance through the Octane benchmark as well. Figure 13 presents the performance comparison among the original v8, v8 with SDCG, and v8 with `libmpk`. Note that originally, v8 has not deployed W⊕X to protect its code cache so far. Our approach only introduces 0.81% overall performance loss, compared with 6.68% caused by SDCG.

To summarize, our `libmpk`-based approaches, which are free of the race condition attacks, outperform the `mprotect()`-based approach currently applied in practice to enforce W⊕X protection on code cache pages with negligible overhead.

In-memory key-value store. To study the performance overhead of `libmpk` when protecting large memory, we evaluate the modified Memcached whose key-value pairs are isolated by `libmpk`. More specifically, the modified Memcached preallocates 1 GB memory, which is used instead of slab pages allocated by glibc `malloc()` to store key-value pairs. Besides the original Memcached, we also evaluate the Memcached whose key-value pairs are protected by `mprotect()`. To study the performance of `mpk_mprotect()` in real-world applications, we also create the Memcached guarded by `libmpk` with permission synchronized as another evaluation target for comparison. Each aforementioned version of Memcached launches with four concurrent threads, and we connect to it remotely through `twemperf` [34]. We create from 250 to 1,000 connections per second, and 10 requests are sent during each connection.

Figure 14 presents the evaluation results. The modified Memcached hardened by `libmpk` only has 0.01% overhead in terms of data throughput and almost no overhead regarding concurrent connections processed per second, which indicates that `libmpk` performs well even when protecting a huge number of pages. By contrast, `mprotect()` introduces nearly 89.56% overhead in terms of data throughput when protecting 1 GB memory in Memcached and a large number of unhandled concurrent connections accumulate in this case. This is because `mprotect()` involves page table traversing, which is considered expensive when dealing with a large number of pages. To evaluate the synchronization service of `libmpk` in practice, we also run Memcached protected by `mpk_mprotect()`. This design ensures the same semantics but outperforms `mprotect()` 8.1× regarding throughput.

`libmpk` provides the same functionality of `mprotect()` with much better performance when protecting huge memory. Moreover, in multi-threading applications, using `mprotect()` to ensure in-thread memory isolation requires lock, which is not required when using `libmpk` because of its inherent property.

7 Discussion

In this section, we discuss a potential attack on both Intel MPK and `libmpk`.

Rogue data cache load (Meltdown). We found that Intel MPK can suffer from the rogue data cache load, also known as the Meltdown attack [19,24]. The Meltdown attack is possible because current Intel CPUs check the access permission to a specific memory page after they have loaded it into the cache. MPK is not an exception because Intel CPUs check the access rights of `PKRU` when checking the page permission at the same pipeline phase. This allows attackers to infer the content of a present (accessible) page even when its protection key has no access right. Since Intel is considering hardware-level mitigation techniques [19], we believe this problem will be solved in the near future.

8 Related Work

MPK applications. While conducting our study, we noticed that there were a few ongoing studies using MPK to achieve different goals. Burow et al. [8] leverage both MPK

and memory protection extension (MPX) to efficiently isolate the shadow stack. ERIM [35] utilizes MPK to isolate sensitive code and data. MemSentry [21] provides a unified memory isolation framework to use various hardware features, including MPK and Memory Protection Extensions (MPX), with the same interface. XOM-Switch [39] relies on MPK to enable execute-only memory for unmodified binaries, and IskiOS [16] leverages MPK and kernel page table isolation (KPTI) to enforce execute-only memory in kernel. Our effort to provide a software abstraction for MPK is orthogonal to these studies, which are all potential applications of `libmpk`. These schemes can leverage `libmpk` to achieve secure and scalable key management to create as many sensitive memory regions as required securely.

Memory protection with other hardware features. Other hardware features exist for efficient memory protection such as ARM Domain [5] and IBM Storage Protection [18], which have a similar concept to MPK. For instance, ARMlock [40], FlexDroid [31], and Shreds [9] rely on Domain to isolate untrusted program modules, third-party libraries, and sensitive code modules, respectively. `libmpk` helps to port these applications from ARM to the Intel platform.

Software-based fault isolation (SFI). SFI [36] prohibits unintended memory accesses by inserting address masking instructions just before load and store instructions. This idea motivates many applications to utilize and further optimize it. Sandboxing mechanisms, such as Native Client (NaCl) [14, 30], relies on SFI to isolate untrusted code. Code-Pointer Integrity [23] also uses SFI to protect the code pointers from unsanitized memory accesses. SFI enables an application to partition its memory into multiple regions, but the cost of address masking limits the shape of partitions, which are commonly contiguous pieces of memory. By contrast, MPK enables an application to partition the memory into the regions with arbitrary shape. Further, the overhead of SFI on address masking increases by the number of isolated memory regions, unlike MPK.

Multiple virtual address spaces. Using multiple virtual address spaces for a single program can protect the memory of sensitive or untrusted components from the others. Some systems [7, 17, 29, 37] rely on multiple page tables to isolate the memory of threads in a single process from each other. Other systems [6, 11, 25] also provide different memory views to individual threads or small execution units using separated page tables. Kenali [32] uses a page-table-based isolation mechanism to protect sensitive data in which a separate page table is created for each thread. Unlike `libmpk`, these mechanisms suffer from non-negligible performance overhead resulting from slow and frequent page table switches.

9 Conclusion

Intel MPK supports efficient per-thread permission control on groups of pages. However, its hardware implementation and software support suffer from security, scalability, and semantic-gap problems. `libmpk` proposes a secure, scalable, and semantic-gap-mitigated software abstraction of MPK for developers to perform fast memory protection and domain-based isolation in their applications. Evaluation results show that `libmpk` incurs negligible performance overhead (<1%) for domain-based isolation and better performance for a substitute of `mprotect()` when adopted to real-world applications: OpenSSL, JavaScript JIT compiler, and Memcached.

10 Acknowledgment

We thank the anonymous reviewers, and our shepherd, John Criswell, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] "Exec Shield", new Linux security feature, 2003. https://lwn.net/Articles/31032/.

[2] Linux kernel, v4.20, 2018. https://elixir.bootlin.com/linux/v4.20-rc1/source/mm/mprotect.c#L630.

[3] Pkeys(7) linux programmer's manual, 2018. http://man7.org/linux/man-pages/man7/pkeys.7.html.

[4] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[5] ARM. ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition, 2018.

[6] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, April 2008.

[7] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium (Security)*, San Diego, CA, August 2003.

[8] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[9] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained Execution

Units with Private Memory. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[10] Gustavo Duarte. How the Kernel Manages Your Memory, 2009. https://manybutfinite.com/post/how-the-kernel-manages-your-memory/.

[11] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[12] Apache Software Foundation. Apache HTTP Server Project, 2018. https://httpd.apache.org/.

[13] Free Software Foundation. The gnu c library, 2018. https://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory-Protection.

[14] Google. NaCl SFI model on x86-64 systems. https://developer.chrome.com/native-client/reference/sandbox_internals/x86-64-sandbox.

[15] Google. The JavaScript Benchmark Suite for the modern web, 2017. https://developers.google.com/octane.

[16] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. IskiOS: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[17] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[18] IBM. Power ISATM Version 3.0 B, 2017.

[19] Intel. Intel Analysis of Speculative Execution Side Channels, 2018.

[20] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2018.

[21] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[22] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[23] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[24] 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 (Security)*, Baltimore, MD, August 2018.

[25] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[26] N. Mehta and Codenomicon. The Heartbleed Bug, 2014. http://heartbleed.com/.

[27] Microsoft. ChakraCore is the core part of the Chakra Javascript engine that powers Microsoft Edge, 2018. https://github.com/Microsoft/ChakraCore.

[28] Mozilla. Spidermonkey, 2018. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[29] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, August 2003.

[30] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, August 2010.

[31] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing In-App Privilege Separation in Android. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[32] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[33] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and Protecting Dynamic Code Generation. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[34] Twitter. twemperf, 2018. https://github.com/twitter-archive/twemperf.

[35] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[36] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, December 1993.

[37] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[38] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[39] Mingwei Zhang, Ravi Sahita, and Daiping Liu. eXecutable-Only-Memory-Switch (XOM-Switch): Hiding Your Code From Advanced Code Reuse Attacks in One Shot. In *Black Hat Asia Briefings (Black Hat Asia)*, Singapore, March 2018.

[40] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers

Jia-Ju Bai
Tsinghua University

Julia Lawall
Sorbonne University/Inria/LIP6

Qiu-Liang Chen, Shi-Min Hu
Tsinghua University

Abstract

In Linux device drivers, use-after-free (UAF) bugs can cause system crashes and serious security problems. According to our study of Linux kernel commits, 42% of the driver commits fixing use-after-free bugs involve driver concurrency. We refer to these use-after-free bugs as *concurrency use-after-free bugs*. Due to the non-determinism of concurrent execution, concurrency use-after-free bugs are often more difficult to reproduce and detect than sequential use-after-free bugs.

In this paper, we propose a practical static analysis approach named DCUAF, to effectively detect concurrency use-after-free bugs in Linux device drivers. DCUAF combines a local analysis analyzing the source code of each driver with a global analysis statistically analyzing the local results of all drivers, forming a *local-global analysis*, to extract the pairs of driver interface functions that may be concurrently executed. Then, with these pairs, DCUAF performs a *summary-based lockset analysis* to detect concurrency use-after-free bugs. We have evaluated DCUAF on the driver code of Linux 4.19, and found 640 real concurrency use-after-free bugs. We have randomly selected 130 of the real bugs and reported them to Linux kernel developers, and 95 have been confirmed.

1 Introduction

Use-after-free (UAF) bugs in device drivers are often dangerous. They not only cause system crashes, but also can be exploited by hackers to attack the operating system [7, 39, 40]. Among use-after-free bugs, *concurrency use-after-free bugs*, which are due to concurrent execution, are more difficult to detect. Indeed, they are not always triggered at runtime due to the non-determinism of concurrent execution. According to our study of Linux kernel commits, 42% of the driver commits fixing use-after-free bugs involve driver concurrency, and nearly all of these concurrency use-after-free bugs appear to have been found by manual inspection or runtime testing.

To detect use-after-free bugs, many approaches use dynamic analysis [6, 25, 33, 36, 45] to monitor memory accesses at runtime. However, the code coverage and detection results of these approaches heavily rely on the tested workloads. Several approaches [41, 42, 44] use static analysis to detect use-after-free bugs. They can cover much code and find many possible bugs without running the tested programs. However, these approaches are designed to detect use-after-free bugs that occur within sequential execution instead of those due to concurrency. Some static approaches [12, 13, 17, 37, 38] for detecting data races in device drivers can find concurrency use-after-free bugs. However, when identifying which driver functions may be concurrently executed, they assume that all driver interface functions can be concurrently executed [13, 37, 38] or rely on manual guidance [12, 17]. These strategies can introduce many false positives or require much manual work. They often report many data races, but many of the reported races are benign or false positives, and only a few are real concurrency use-after-free bugs.

In this paper, we propose DCUAF, a static analysis approach to detect concurrency use-after-free bugs in Linux device drivers. DCUAF first uses a local-global strategy to extract *concurrent function pairs*, namely the pairs of driver interface functions that can be executed concurrently. Then, with these function pairs, DCUAF performs a summary-based lockset analysis to detect concurrency use-after-free bugs. Our local-global strategy has two stages. In the local stage, DCUAF scans the code of each driver, and identifies calls to lock-acquiring functions, such as `spin_lock`. According to these calls and the driver's function call graph, DCUAF extracts *local concurrent interface pairs*, namely the pairs of driver interfaces that may be concurrently executed for the driver. In the global stage, DCUAF gathers the local concurrent interface pairs of all drivers and performs a statistical analysis to identify the pairs of driver interfaces that are frequently considered to be concurrently executed, from which it produces *global concurrent interface pairs*. Using these interface pairs, for each driver, DCUAF identifies the driver interface functions associated with these pairs as concurrent function pairs for this driver. For each driver function in a concurrent function pair, our lockset analysis analyzes each

variable access in the driver function, and records the lockset that protects this access. Then, for each pair of accesses in the functions of a concurrent function pair, the analysis compares their variables and locksets, and reports concurrency use-after-free bugs. To improve accuracy, our lockset analysis is inter-procedural, context-sensitive and flow-sensitive, and it maintains function summaries to reduce repeated analysis.

We have implemented DCUAF using Clang 6.0 [9] for Linux drivers. DCUAF is fully automatic, given the set of driver source files in the kernel. Overall, we make four main contributions:

- We perform a study of Linux kernel commits, and find that 42% of driver commits fixing use-after-free bugs involve concurrency. Moreover, we infer that nearly all of these concurrency use-after-free bugs have been found by manual inspection or runtime testing. To find more concurrency use-after-free bugs in device drivers, we propose to explore static analysis.
- We propose DCUAF, to detect concurrency use-after-free bugs in device drivers. To our knowledge, DCUAF is the first systematic static approach that targets concurrency use-after-free bugs in device drivers.
- We propose a novel local-global strategy to extract concurrent function pairs.
- We evaluate DCUAF on device drivers in Linux 3.14 and 4.19, and find 559 and 679 concurrency use-after-free bugs, respectively. We manually check these bugs, and find that 526 and 640 bugs are real, respectively. 35 of the real bugs found in Linux 3.14 have been fixed in Linux 4.19. We have randomly selected 130 of the real bugs in Linux 4.19, and reported them to Linux kernel developers. 95 of these bugs have been confirmed.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 shows the challenges of detecting concurrency use-after-free bugs in Linux device drivers and our key techniques to address these challenges. Section 4 introduces DCUAF. Section 5 presents the evaluation. Section 6 discusses how to apply our approach to other kinds of driver problems. Section 7 gives related work, and Section 8 concludes.

2 Background

We first introduce the Linux driver interface model, and then motivate our work by a concurrency use-after-free bug in a Linux driver and by our study of Linux kernel commits.

2.1 Linux Driver Interface Model

A Linux device driver needs to implement some specified *driver interfaces*, including *kernel-driver interfaces* and *interrupt handler interfaces*. A kernel-driver interface is invoked by non-driver threads through function calls when the driver communicates with related management code in the kernel, and an interrupt handling interface is called when a hardware interrupt occurs. We call the driver functions implemented for these driver interfaces *driver interface functions*. The driver interface functions are assigned to driver interfaces through specific data structure fields or specific kernel interfaces. The driver interface functions form the entry points of the driver, so all other functions defined in the driver are called by them. From the kernel's point of view, different drivers of the same device class should have the same functionalities, so drivers in the same device class share the same driver interfaces.

Figure 1 shows two typical Ethernet controller drivers (*dl2k* and *ne2k-pci*) in Linux 4.19. These drivers both define a net_device_ops data structure, containing some function pointer fields. Each network controller driver uses this data structure to communicate with network management code in the kernel, and each function pointer field represents a kernel-driver interface that performs a specific functionality of the driver. For example, in the net_device_ops data structure, the field `ndo_open` is used to open a network device, the field `ndo_stop` is used to close a network device, and the field `ndo_start_xmit` is used to transmit data packets. These drivers also both call a kernel interface `request_irq`, to register their interrupt handler functions through a function pointer argument. According to the functionalities of these driver interfaces, for a given network device instance, the interface net_device_ops.`ndo_start_xmit` can be concurrently executed with the interrupt handler function, but the interface net_device_ops.`ndo_open` is never concurrently executed with the interface net_device_ops.`ndo_stop`. However, whether two driver interfaces can be concurrently executed is often poorly documented in the Linux kernel.

Based on this driver interface model, *driver concurrency is often determined by the concurrent execution of driver interfaces.* Thus, to detect concurrency problems in device drivers, we need to know which driver interfaces can be concurrently executed. We refer to driver interfaces that can be concurrently executed as *concurrent interface pairs*.

2.2 Concurrency Use-After-Free Bug

Use-after-free (UAF) bugs are known to be hard to debug. If the freed memory is not reallocated, and thus not reinitialized, before the use, there is no visible problem, so the bug can linger. If the freed memory is reallocated and reinitialized before the use, then a read use can return an unexpected value and a write use can destroy data relied on by another part of the program. These problems are compounded in the case of kernel code, as the memory can be reinitialized by a different process, allowing information leaks. Some recent works [39, 40] have discussed how to exploit use-after-free bugs to attack an operating system.

```
FILE: linux-4.19/drivers/net/ethernet/dlink/dl2k.c
 98. static const struct net_device_ops netdev_ops = {
 99.     .ndo_open = rio_open,
100.     .ndo_stop = rio_close,
101.     .ndo_start_xmit = start_xmit,
         ......
108. };
- - - - - - - - - - - - - - - - - - - - - - - - - -
628. static int rio_open(...) {
         ......
640.     err = request_irq(irq, rio_interrupt, ...);
         ......
655. }
```

```
FILE: linux-4.19/drivers/net/ethernet/8390/ne2k-pci.c
203. static const struct net_device_ops ne2k_netdev_ops = {
204.     .ndo_open = ne2k_pci_open,
205.     .ndo_stop = ne2k_pci_close,
206.     .ndo_start_xmit = ei_start_xmit,
         ......
215. };
- - - - - - - - - - - - - - - - - - - - - - - - - -
432. static int ne2k_pci_open(...) {
         ......
434.     int ret = request_irq(dev->irq, ei_interrupt, ...);
         ......
443. }
```

Figure 1: Examples of driver interfaces.

We motivate our work by a real concurrency use-after-free bug in the Linux *cw1200* wireless controller driver. The *cw1200* driver manages the ST-Ericsson CW1200 wireless controller that is used in many embedded systems. The bug was introduced in Linux 3.11 (Sep. 2013) and was fixed 5 years later (Dec. 2018) by us, based on a report generated by DCUAF. Figure 2 shows the driver code related to this bug in Linux 4.19. In the ieee80211_ops data structure, as the driver interfaces represented by the fields `hw_scan` and `bss_info_changed` can be executed concurrently, the driver interface functions `cw1200_hw_scan` and `cw1200_bss_info_changed` can be executed concurrently. In *scan.c*, the function `cw1200_hw_scan` calls `dev_kfree_skb` to free `frame.skb` on line 126, without holding the lock `priv->conf_mutex`. In *sta.c*, the function `cw1200_bss_info_changed` calls `cw1200_upload_beacon`, which reads `frame.skb` on line 2221, while holding the lock `priv->conf_mutex`. Because the free operation is performed without holding a lock but the read operation is performed while holding a lock, a concurrency use-after-free bug may occur. To fix this bug, our commit [2] moved the call to `mutex_unlock` in `cw1200_hw_scan` behind the call to `dev_kfree_skb`.

This example illustrates some reasons why concurrency use-after-free bugs occur in device drivers: (1) Determining which driver interfaces can be executed concurrently requires substantial driver knowledge. In the example, without knowing wireless controller drivers in the Linux kernel well, it may be hard to know that the driver interfaces represented by the fields `hw_scan` and `bss_info_changed` can be executed concurrently. (2) Concurrency use-after-free bugs are not always

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/main.c
208. static const struct ieee80211_ops cw1200_ops = {
         ......
215.     .hw_scan = cw1200_hw_scan,
         ......
223.     .bss_info_changed = cw1200_bss_info_changed,
         ......
238. };
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/scan.c
 54. int cw1200_hw_scan(...) {
         ......
 91.     mutex_lock(&priv->conf_mutex);
         ......
123.     mutex_unlock(&priv->conf_mutex);
125.     if (frame.skb)
126.         dev_kfree_skb(frame.skb); // FREE
         ......
129. }
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/sta.c
1799. void cw1200_bss_info_changed(...) {
          ......
1807.     mutex_lock(&priv->conf_mutex);
          ......
1849.     cw1200_upload_beacon(...);
          ......
2075.     mutex_unlock(&priv->conf_mutex);
          ......
2081. }
- - - - - - - - - - - - - - - - - - - - - - - - - -
2189. static int cw1200_upload_beacon(...) {
          ......
2221.     mgmt = (void *)frame.skb->data; // READ
          ......
2238. }
```

Figure 2: A reported bug in the *cw1200* driver in Linux 4.19.

triggered in real execution and are hard to reproduce. In the example, `cw1200_hw_scan` and `cw1200_bss_info_changed` are not always concurrently executed at runtime. (3) Multiple functions needs to be considered, including the concurrently executed driver interface functions and the functions they call. In the example, three driver functions are involved.

2.3 Our Study of Linux Kernel Commits

To understand the state of the art in detecting use-after-free bugs in the Linux kernel, we study the Linux kernel commits to the mainline kernel [27]. We select the non-merge commits from Jan. 2016 to Dec. 2018 that fix use-after-free bugs, by searching (`git --grep`) for "use after free"and "use-after-free" in the log message, resulting in 949 commits. From them, we identify the driver commits, i.e., those affecting the *drivers* or *sound* directories. For the driver commits, we then study the log messages and code changes to determine: (1) whether the reported bugs are concurrency use-after-free bugs; (2) whether the reported bugs were detected by tools. Table 1 shows the results.

As shown in the table, 49% of the use-after-free related commits are for device drivers. Moreover, 42% of the use-after-free driver commits involve concurrency. Around 65% of the driver use-after-free commits, whether or not they involve concurrency, mention the use of tools, including KASAN [19], Syzkaller [34], Coverity [11], Coccinelle [30] and LDV [24].

Time	Commits	Drivers	Concurrency	Tool
2016 (Jan-Dec)	186	111	42 (38%)	26
2017 (Jan-Dec)	478	205	87 (42%)	49
2018 (Jan-Dec)	285	145	66 (46%)	52
Total	949	461	195 (42%)	127

Table 1: Linux kernel commits fixing use-after-free bugs.

Tool	KASAN	Syzkaller	Coverity	Coccinelle	LDV
Type	Runtime	Runtime	Static	Static	Static
Commit	92	28	4	2	1
Concurrency	38	18	0	0	0

Table 2: Use-after-free bugs found by different analyses and testing tools.

For the remaining driver use-after-free commits, we infer that the reported bugs in these commits are found by manual inspection of the source code and execution failures.

Table 2 breaks down the tool results by the specific tools. KASAN and Syzkaller are runtime testing tools. 120 of the commits fix the bugs found by these tools, including 56 that fix concurrency use-after-free bugs. Coverity, Coccinelle and LDV are static analysis tools. Only 7 of the commits fix bugs found by these tools, with no bug involving concurrency.

From the results, we can infer that nearly all of reported use-after-free bugs in released kernels have been found by manual inspection or runtime testing. However, runtime testing heavily relies on workloads to cover code, and thus it may miss many real bugs in practice. For this reason, it is important to explore static analysis as an alternative to detect concurrency use-after-free bugs in device drivers, but no systematic static tool has yet been proposed. Thus, we aim to design an effective static approach to solve this problem.

3 Challenges and Key Techniques

Our basic idea is to first extract *concurrent function pairs*, namely the pairs of driver interface functions that can be executed concurrently, and then perform a lockset analysis on these pairs of functions to detect concurrency use-after-free bugs. Implementing this idea requires addressing two main challenges:

C1: Extracting concurrent function pairs. Determining which driver functions may be concurrently executed requires substantial driver knowledge. Moreover, the Linux kernel documentation often lacks explicit descriptions about the concurrency of driver interfaces, and thus driver developers may err when implementing the code.

C2: Accuracy and efficiency of code analysis. The Linux driver code base is very large, amounting to 12.6M code lines in our tested version Linux 4.19. Thus, the lockset analysis can be quite time-consuming.

To solve the above challenges, we propose two key techniques. For *C1*, we propose a local-global strategy to extract concurrent function pairs from driver source files in the kernel. For *C2*, we propose a summary-based lockset analysis to detect concurrency use-after-free bugs.

3.1 Local-Global Strategy

Reviewing the example in Figure 2 suggests the following strategy: *concurrent interface pairs could be inferred according to the lock-acquiring function calls in driver interface functions.* In Figure 2, `cw1200_bss_info_changed` and `cw1200_hw_scan` both call `mutex_lock` with a lock variable `priv->conf_mutex`. This information suggests that these two driver functions may be concurrently executed. From this information, we could infer that the related driver interfaces `hw_scan` and `bss_info_changed` in the data structure ieee80211_ops may be a concurrent interface pair. But this kind of inference can be wrong in two common cases:

Case 1. It is possible that two functions that acquire the same lock are actually never concurrently executed. Figure 3 shows an example in the `e100` Ethernet controller driver. The driver functions `e100_enable_irq` and `e100_disable_irq` both call the lock-acquiring function `spin_lock_irqsave` with the same lock variable `nic->cmd_lock`, but they are also both called by the driver function `e100_netpoll`. This suggests that the spinlock acquired in `e100_enable_irq` and `e100_disable_irq` is used by `e100_netpoll` to synchronize with other driver functions, not to synchronize the calls to these two functions with each other.

```
FILE: linux-4.19/drivers/net/ethernet/intel/e100.c
616. static void e100_enable_irq(...) {
         ......
620.     spin_lock_irqsave(&nic->cmd_lock, flags);
         ......
623.     spin_unlock_irqsave(&nic->cmd_lock, flags);
624. };

626. static void e100_disable_irq(...) {
         ......
630.     spin_lock_irqsave(&nic->cmd_lock, flags);
         ......
633.     spin_unlock_irqsave(&nic->cmd_lock, flags);
634. };
-----------------------------------------------
2238. static void e100_netpoll(...) {
          ......
2242.     e100_disable_irq();
          ......
2245.     e100_enable_irq();
2246. }
```

Figure 3: Part of the *e100* driver in Linux 4.19.

Case 2. For two given driver interfaces, only a few of the drivers having the both driver interfaces acquire the same lock in these two driver interfaces, but most drivers do not. Table 3 shows some examples. The first and second columns show the names of involved driver interfaces; the third column shows the number of driver source files that have both

Driver Interface 1	Driver Interface 2	Both	Concurrent
spi_driver.probe	spi_driver.remove	227	3
file_operations.open	file_operations.llseek	462	3
watchdog_ops.start	watchdog_ops.stop	75	1
net_device_ops.ndo_open	ethtool_ops.get_link	124	2

Table 3: Example drivers having the same driver interfaces.

the involved driver interfaces; the fourth column shows the number of driver source files where the involved driver interfaces both acquire the same lock. For example, 227 driver source files have the driver interfaces spi_driver.`probe` and spi_driver.`remove`, but only 3 acquire the same lock in both driver interfaces. Indeed, spi_driver.`probe` is used to initialize an SPI device while spi_driver.`remove` is used to remove a running SPI device, and a device cannot be initialized and removed at the same time. Thus, the two driver interfaces should not be concurrently executed.

To handle the above two cases, we collect information about the lock usage of each driver as *local* information, and then combine the information about all drivers to perform a *global* statistical analysis. Based on this idea, we propose a *local-global strategy* to extract concurrent function pairs from driver code. The local and global stages handle *Case 1* and *Case 2*, respectively.

Local stage. In this stage, our strategy analyzes each driver source file, and extracts *local concurrent interface pairs*, namely the pairs of driver interfaces that may be concurrently executed for each driver. Figure 4 defines this stage, which has three steps:

Step 1. This step identifies the pairs of possible concurrently executed functions in each driver source file. Firstly, this step clears the result set *pos_func_pair_set*, and collects the set of lock-acquiring function calls as the set *lock_call_-set* (lines 1-2). Secondly, this step performs an alias analysis and checks each call in the set *lock_call_set* (lines 4-15). We identify whether the locks are the same by checking whether the related lock variables are aliased. If two different calls in the set have an aliased lock variable, their callers are considered as a pair of possible concurrently executed functions for the source file. In this case, the pair of callers is added to *pos_func_pair_set*. Finally, this step returns the final value of *pos_func_pair_set* (line 17).

Our alias analysis is field-based [16] and focuses on the lock variables stored in data structure fields. We take this strategy for two reasons. Firstly, drivers often use data structure fields to share data (such as locks) between different functions, as illustrated in Figures 2 and 3. Secondly, a variable stored in a data structure field can be explicitly distinguished from other variables using the data structure type and field name. However, for some lock frameworks, their lock-acquiring functions do not have any argument, such as `rcu_read_lock`. Thus, our analysis does not support these lock frameworks at present.

Step 2. This step filters out the pairs of possible concurrently executed functions that may actually not be executed concurrently. For each pair of possible concurrently executed functions, this step collects and checks the sets of their ancestors in the call graph (lines 2-5). Note that "ancestor" here include callers, callers of the callers, etc. Common ancestors are only collected up to the point of encountering a function that has no caller in the driver but instead is only assigned to a function pointer. As described in Section 2.1, a driver interface that forms an entry point of the driver is often presented as a function pointer stored in a data structure field. If the two driver functions have a common ancestor, the pair of the two functions is deleted from the set *pos_func_pair_set* (line 6), to avoid the possible false positives exemplified by *Case 1*. Finally, this step returns *pos_func_pair_set* (line 9).

```
Step 1: Get the pairs of possible concurrently executed functions
 1: pos_func_pair_set := ø;
 2: lock_call_set := GetLockCall();
 3: for i := 0 to SizeOf(lock_call_set) - 1 do
 4:     lock_call1 := lock_call_set[i];
 5:     lock_var1 := GetLockVar(lock_call1);
 6:     caller_func1 := GetCallerFunc(lock_call1);
 7:     for j := i + 1 to SizeOf(lock_call_set) - 1 do
 8:         lock_call2 := lock_call_set[j];
 9:         lock_var2 := GetLockVar(lock_call2);
10:         caller_func2 := GetCallerFunc(lock_call2);
11:         if lock_var1 is aliased to lock_var2 then
12:             func_pair := <caller_func1, caller_func2>
13:             Add func_pair to pos_func_pair_set;
14:         end if
15:     end for
16: end for
17: return pos_func_pair_set;

Step 2: Filter out may-false pairs of concurrently executed functions
 1: foreach func_pair in pos_func_pair_set do
 2:     <caller_func1, caller_func2> := GetFuncPair(func_pair);
 3:     func_set1 := GetAncestorFunc(caller_func1);
 4:     func_set2 := GetAncestorFunc(caller_func2);
 5:     if func_set1 ∩ func_set2 ≠ ø then
 6:         Delete func_pair from pos_func_pair_set;
 7:     end if
 8: end foreach
 9: return pos_func_pair_set;

Step 3: Get local concurrent interface pairs
 1: local_interface_pair_set := ø;
 2: foreach func_pair in pos_func_pair_set do
 3:     <caller_func1, caller_func2> := GetFuncPair(func_pair);
 4:     interface_set1 := GetDriverInterface(caller_func1);
 5:     interface_set2 := GetDriverInterface(caller_func2);
 6:     foreach interface1 in interface_set1 do
 7:         foreach interface2 in interface_set2 do
 8:             if interface1 == interface2 then
 9:                 continue;
10:             end if
11:             interface_pair := <interface1, interface 2>
12:             Add interface_pair to local_interface_pair_set;
13:         end foreach
14:     end foreach
15: end foreach
16: return local_interface_pair_set;
```

Figure 4: The local stage.

Step 3. From the remaining pairs of possible concurrently executed functions, this step extracts the local concurrent interface pairs for the driver. For each function in a pair of possible concurrently executed functions, this step gets the set of driver interfaces that call this function (lines 3-5). Then, this step computes the Cartesian product of the two sets of driver interfaces, omitting the pairs where both driver interfaces are the same (lines 6-14) to avoid the case that different driver functions are assigned to the same driver interface.

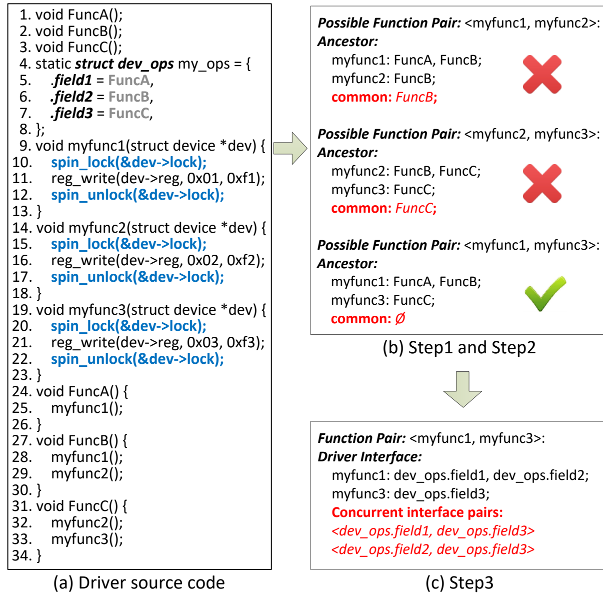


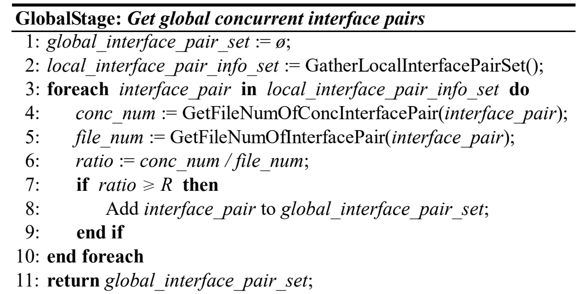
Figure 5: Example of performing the local stage.

Example. To illustrate the local stage, we use some driver-like code shown in Figure 5. In Figure 5(a), the data structure dev_ops has three fields `field1`, `field2` and `field3`, and each of them stores a driver function, namely `FuncA`, `FuncB` and `FuncC`. *Step 1* identifies three lock-acquiring function calls on lines 10, 15 and 20, with the same lock `dev->lock`. Thus, this step gets three pairs of possible concurrently executed functions, namely `<myfunc1, myfunc2>`, `<myfunc2, myfunc3>` and `<myfunc1, myfunc3>`. Then, as shown in Figure 5(b), for each pair, *Step 2* checks the ancestors of the involved functions. `myfunc1` and `myfunc2` have a common caller `FuncB`, and `myfunc2` and `myfunc3` have a common caller `FuncB`, and thus the two pairs are filtered out, leaving only `<myfunc1, myfunc3>`. Finally, as shown in Figure 5(c), *Step 3* uses this function pair to get two lock concurrent interface pairs.

Note that the local stage assumes that a driver function or interface cannot be concurrently executed with itself. The main reason is that only analyzing the lock usage in a function is insufficient to infer whether this function can be concurrently executed with itself. However, this case indeed exists for some drivers, and may make our strategy miss some real local concurrent interface pairs.

Global stage. In this stage, with the local concurrent interface pairs of each driver, we perform a statistical analysis to extract *global concurrent interface pairs* for all drivers.

As shown in Figure 6, this stage first clears the result set *global_interface_pair_set*, and gathers the local concurrent interface pairs of all drivers (lines 1-2). Secondly, this stage handles each concurrent interface pair *interface_pair* in the gathered set (lines 4-9). It calculates the percentage of source files containing the two driver interfaces that have the two driver interfaces as an extracted local concurrent interface pair. This percentage is represent as *ratio* in Figure 6. If *ratio* $\geqslant R$ (a given threshold), *interface_pair* is considered as a global concurrent interface pair, and is added to the set *global_interface_pair_set*. Finally, this stage returns the final value of *global_interface_pair_set* (line 11).



GlobalStage: ***Get global concurrent interface pairs***

```
1: global_interface_pair_set := ø;
2: local_interface_pair_info_set := GatherLocalInterfacePairSet();
3: foreach interface_pair in local_interface_pair_info_set do
4:     conc_num := GetFileNumOfConcInterfacePair(interface_pair);
5:     file_num := GetFileNumOfInterfacePair(interface_pair);
6:     ratio := conc_num / file_num;
7:     if ratio ⩾ R then
8:         Add interface_pair to global_interface_pair_set;
9:     end if
10: end foreach
11: return global_interface_pair_set;
```

Figure 6: The global stage.

In this stage, the value of the threshold R is important, because the number of extracted global concurrent interface pairs decreases as R becomes larger. Increasing R may cause more false global concurrent interface pairs to be dropped, but more real interface pairs may be missed. We study the impact of the value of R in Section 5.3.

With the extracted global concurrent interface pairs, we identify concurrent function pairs for each driver. Specifically, given two driver interfaces in a driver, if they are in a global concurrent interface pair, the two driver functions associated with these driver interfaces are identified as a concurrent function pair for this driver.

3.2 Summary-Based Lockset Analysis

To improve accuracy and efficiency, our summary-based lockset analysis has the following properties: (1) The analysis is context-sensitive and inter-procedural, in order to maintain locksets and detect bugs across functions calls. (2) The analysis is flow-sensitive to improve accuracy. (3) The analysis uses function summaries to reduce repeated analysis and improve efficiency. (4) The analysis is field-based, and it focuses on the variables stored in data structure fields.

Given driver source code and a concurrent function pair, our lockset analysis has two steps:

Step 1. For each driver function in the concurrent function pair, this step collects the lockset of each variable access (read or write). During the collection, this step uses function summaries to handle called driver functions. Each function summary has the function name, source file name and a set that stores the information about all variable accesses in the function, including the accessed variable, the lockset of the access, the code path that reaches the access from the start of the function and the location of the access.

Figure 7 shows the treatment of a called function *func* by the caller *caller*, with *caller*'s function summary *caller_sum*, the collected lockset *lockset_caller* and the code path *path_info_caller* through *caller* when reaching the call to *func*. Firstly, this step checks whether there is already a call to *func* in the current path by searching *path_info_caller* (lines 1-3). If so, this step returns to avoid infinite looping on recursive calls. Secondly, this step searches the stored function summaries to check whether *func* has been handled (line 4). If so, this step directly uses its function summary *func_sum*. Otherwise, this step performs flow-sensitive analysis to collect information about variable accesses in *func* and then stores *func*'s function summary *func_sum* (lines 5-8). Thirdly, whether an existing function summary is found or a new one is created, this step gets the set *access_info_set* that stores the information about all variable accesses in *func* (line 9). Fourthly, for each variable access in *access_info_set*, this step concatenates its lockset and code path to the end of the *caller*'s lockset and code path, and then stores the information about this variable access in the function summary *caller_sum* (lines 10-16). Using function summaries, repeated flow-sensitive analyses of function definitions are reduced, which can improve the efficiency of the lockset analysis.

```
HandleFunc(func, caller_sum, lockset_caller, path_info_caller)
1: if func exists in path_info_caller then
2:     return;
3: end if
4: func_sum := FindFuncSummary(func);
5: if func_sum == ø then
6:     func_sum := AnalyzeFuncSummary(func);
7:     StoreFuncSummary(func_sum);
8: end if
9: access_info_set := GetAccessInfoSet(func_sum);
10: foreach access_info in access_info_set do
11:     lockset := lockset_caller + GetLockSet(access_info);
12:     path_info := path_info_caller + GetPathInfo(access_info);
13:     SetLockSet(access_info, lockset);
14:     SetPathInfo(access_info, path_info);
15:     AddAccessInfo(access_info, caller_sum);
16: end foreach
```

Figure 7: Handling a called function in our lockset analysis.

Step 2. For each pair of variable accesses in the driver functions of a concurrent function pair, this step compares the accessed variables and held locksets, and reports a concurrency use-after-free bug if: (1) the accessed variables are the same; (2) the intersection of the locksets is empty; (3) one of the accessed variable is used as an argument of a call to a memory freeing function. If both of the accessed variables are used as an argument of a call to a memory freeing function, a double-free bug is also reported.

4 Approach

Based on the two key techniques in Section 3, we propose a practical static approach named DCUAF, to detect concurrency use-after-free bugs in Linux device drivers. We implement DCUAF using Clang 6.0 [9], and perform static analysis on the LLVM bytecode of the driver code. Figure 8 shows the overall architecture of DCUAF.

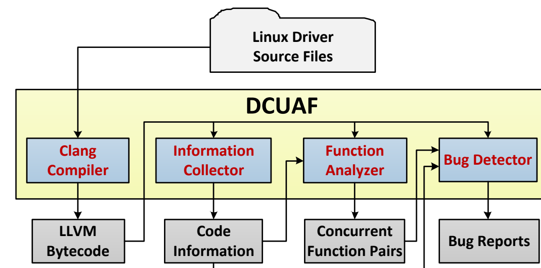


Figure 8: Overall architecture of DCUAF.

Based on this architecture, DCUAF has four phases:

P1: Source code compilation. In this phase, the *Clang compiler* compiles the driver source files and generates their LLVM bytecode files. Because a driver can be implemented across multiple source files, a driver function can call another driver function that is defined in another source file. During linking, DCUAF thus records the set of related source files. This set is used to locate the definition of a called function that is not in the same source file as the caller function.

P2: Code information collection. In this phase, the *information collector* analyzes each LLVM bytecode file, and records code information in a database. The information includes the name and position of each function definition and interrupt handler function, driver functions assigned to function pointers that are stored in data structure fields, the callee and caller functions of each function call, etc. The collected information is used in the remaining phases.

P3: Concurrent function pair extraction. In this phase, with the collected information, the *function analyzer* uses our local-global strategy to analyze LLVM bytecode files. It produces concurrent function pairs for each driver.

P4: Bug detection. In this phase, with the collected code information and extracted concurrent function pairs, the *bug detector* performs our summary-based lockset analysis to analyze each LLVM bytecode file and detect concurrency use-after-free bugs. Some reported bugs may be repeated, when the two bugs are associated with the same driver function and end up at the same variable access but differ in their code paths. Thus, the bug detector also filters out such repeated bug reports, by checking the positions of variable accesses.

Parallelism. The phases P1, P2 and P4 can work on individual LLVM bytecode files independently, and thus they can be parallelized straightforwardly. In P3, the local stage also works on individual LLVM bytecode files, and thus it can be parallelized, too. Only the global stage must be carried out in a single thread to perform the statistical analysis of the collected information. Because this stage does not require processing LLVM bytecode files, it is fast. Thus, overall, DCUAF can greatly benefit from parallelism.

5 Evaluation

We evaluate DCUAF on the source code of Linux device drivers. To cover different kernel versions, we select an old version 3.14 (released in March 2014) and a recent version 4.19 (released in October 2018). Table 4 shows information about the driver code in these kernel versions.

Description	Linux 3.14	Linux 4.19
Release time	March 2014	October 2018
Driver source files (.c)	11.2K	16.6K
Driver source code lines	6.7M	9.5M

Table 4: Properties of the evaluated driver code.

We run the experiments on a Lenovo x86-64 PC with four Intel i5-3470@3.20G processors and 8GB memory. We use the kernel configuration *allyesconfig* to enable all device drivers that can be compiled for the x86 architecture. We compile the driver code using the Clang 6.0 compiler [9]. Because DCUAF can work in parallel, we configure DCUAF to run on 4 threads.

5.1 Extracting Concurrent Function Pairs

DCUAF first uses our local-global strategy to extract concurrent function pairs. In the global strategy, we set $R = 0.2$. Table 5 shows the results for Linux 3.14 and 4.19.

The results show that DCUAF can scale to large code bases. It handles 5.1M and 7.9M source code lines in 7.9K and 13.1K source files in Linux 3.14 and 4.19, respectively, within 20 minutes. The remaining 1.6M and 1.8M source code lines in 3.3K and 3.5K source files in Linux 3.14 and 4.19 are not handled, because they are not enabled by *allyesconfig* for the x86 architecture.

Description		3.14	4.19
Code handling	Handled source files (.c)	7957	13100
	Handle code lines	5.1M	7.9M
Local stage	Dropped function pairs	61.4K	99.8K
	Remaining function pairs	40.7K	67.8K
Global stage	Candidate concurrent interface pairs	7354	11793
	Global concurrent interface pairs	694	1497
	Extracted concurrent function pairs	15.6K	69.5K
Time usage	Code information collection	10m16s	12m20s
	Local stage	4m36s	5m23s
	Global stage	10s	15s
	Total	14m52s	17m58s

Table 5: Results of extracting concurrent function pairs.

The results also show that our local-global strategy is effective in extracting concurrent function pairs. For example, for Linux 4.19, our strategy extracts 1497 global concurrent interface pairs from the 11,793 candidate concurrent interface pairs identified in the local stage. The remaining interface pairs are not extracted, because they are not identified as being able to execute concurrently by our strategy. For example, DCUAF deletes nearly all interface pairs related to driver initialization and removal (like `probe` and `remove`), which cannot run concurrently in real execution.

Table 6 shows a few of the extracted global concurrent interface pairs in Linux 4.19. The first and second columns show the names of the involved driver interfaces; the third column shows the number of driver source files that have both the involved driver interfaces; the fourth column shows the number of driver source files where the local stage identifies the involved driver interfaces as a concurrent interface pair.

Driver Interface 1	Driver Interface 2	Both	Concurrent
tty_operations.write	tty_operations.put_char	14	12
hc_driver.urb_enqueue	hc_driver.endpoint_disable	16	9
ieee80211_ops.bss_info_changed	ieee80211_ops.hw_scan	12	7
uart_ops.set_termios	console.write	21	14
Interrupt handler	snd_pcm_ops.trigger	49	25

Table 6: Examples of global concurrent interface pairs.

5.2 Detecting Bugs

With the extracted concurrent interface pairs, DCUAF runs our summary-based lockset analysis to detect concurrency use-after-free bugs. To validate whether DCUAF can find known bugs, we use it to check Linux 3.14 drivers. To validate whether DCUAF can find new bugs, we use it to check Linux 4.19 drivers. We also manually check all found bugs to validate accuracy. The results are shown in Table 7.

Description		3.14	4.19
Bug detection	Filter repeated	348	390
	Final detected (real / all)	526 / 559	640 / 679
	Double free (real / all)	82 / 89	117 / 132
	Interrupt handler (real / all)	25 / 25	23 / 23
Time usage		8m43s	10m15s

Table 7: Results of detecting bugs.

The results show that DCUAF finds 559 concurrency use-after-free bugs in Linux 3.14. We identify 526 of them as real bugs that are in 108 source files. Among these bugs, 35 have been fixed in Linux 4.19. Thus, DCUAF can find known bugs.

The results show that DCUAF finds 679 concurrency use-after-free bugs in Linux 4.19. We identify 640 of them as real bugs that are in 132 source files. Among these bugs, 372 are also found in Linux 3.14, and thus they have been present for at least 4.5 years. We have randomly selected 130 of these real bugs, and reported them to Linux kernel developers. 95 of them have been confirmed, and 12 of our patches that fix 42 real bugs have been applied (such as commits [1] and [3]) in the kernel code. Thus, DCUAF can find new bugs.

Among the bugs found by DCUAF, many are also double-free bugs. Specifically, DCUAF finds 89 and 132 double-free bugs in Linux 3.14 and 4.19, respectively, and we identify 82 and 89 of them as real bugs. Furthermore, DCUAF finds 25 and 23 bugs that involve interrupt handling in Linux 3.14 and 4.19, respectively, and we identify all of them as real bugs.

Over 60% of the real bugs found by DCUAF are in network, TTY, character and ISDN drivers. Specifically, 359 and 455 of the found real bugs are in these drivers in Linux 3.14 and 4.19, respectively, amounting to 68% and 71% of all found real bugs. Indeed, compared to other drivers, these drivers have more driver functions that can be concurrently executed.

```
FILE: linux-4.19/drivers/usb/host/r8a66597-hcd.c
1885. static int r8a66597_urb_enqueue(...) {
          ......
1895.     spin_lock_irqsave(&r8a66597->lock, flags);
          ......
1905.     if (!hep->hcpriv)  // READ
          ......
1951.     spin_unlock_irqrestore(&r8a66597->lock, flags);
1952.     return ret;
1953. };
-------------------------------------------------------
1980. static void r8a66597_endpoint_disable(...) {
          ......
1995.     kfree(hep->hcpriv);  // FREE
          ......
2000.     spin_lock_irqsave(&r8a66597->lock, flags);
          ......
2010.     spin_unlock_irqrestore(&r8a66597->lock, flags);
2011. }
-------------------------------------------------------
2304. static const struct hc_driver r8a66597_hc_driver = {
          ......
2320.     .urb_enqueue = r8a66597_urb_enqueue,
          ......
2322.     .endpoint_disable = r8a66597_endpoint_disable,
          ......
2336. }

========== BUG REPORT ==========
# [READ] r8a66597_urb_enqueue (drivers/.../r8a66597-hcd.c, LINE 1905)
  [LOCK] r8a66597_urb_enqueue (drivers/.../r8a66597-hcd.c, LINE 1895)
# [FREE] r8a66597_endpoint_disable (drivers/.../r8a66597-hcd.c, LINE 1995)
```

Figure 9: A confirmed bug in the *r8a66597* driver.

Figure 9 shows a new confirmed bug found by DCUAF in the Linux 4.19 *r8a66597* driver. The *r8a66597* driver manages the Renesas R8A66597 USB host controller that is used in many embedded systems with USB ports. In the data structure hc_driver, the driver interfaces represented by the fields `urb_enqueue` and `endpoint_disable` are extracted as a global concurrent interface pair (the second row in Table 6) by our local-global strategy. Accordingly, DCUAF considers that the driver functions `r8a66597_urb_enqueue` and `r8a66597_endpoint_disable` may be concurrently executed. In `r8a66597_endpoint_disable`, the variable `hep->hcpriv` is freed on line 1995 without holding the spinlock `r8a66597->lock`. But in `r8a66597_urb_enqueue`, this variable is read on line 1905 while holding the spinlock `r8a66597->lock`. Thus, a concurrency use-after-free bug may occur. The bug report generated by DCUAF shows the related call paths, including the positions of the free, read and lock-acquiring operations. To fix this bug, we move the call to `spin_lock_irqsave` (line 2000) before the call to `kfree` (line 1995). Our patch[1] making this change has been applied by the Linux kernel maintainers.

The found bug in Figure 9 was introduced in Linux 2.6.22 (Jul. 2007), and had existed for over 11 years. Indeed, the USB related documentations in the Linux kernel [35] does not mention that the driver interfaces `urb_enqueue` and `endpoint_disable` can be concurrently executed.

5.3 Result Variation

As described in Section 3.1, the value of R is important. The above results are obtained with $R = 0.2$. To see the variation caused by R, we test $R = 0.1, 0.2, 0.3, 0.4$ and 0.5 on the Linux 4.19 drivers. Figure 10 shows the results.

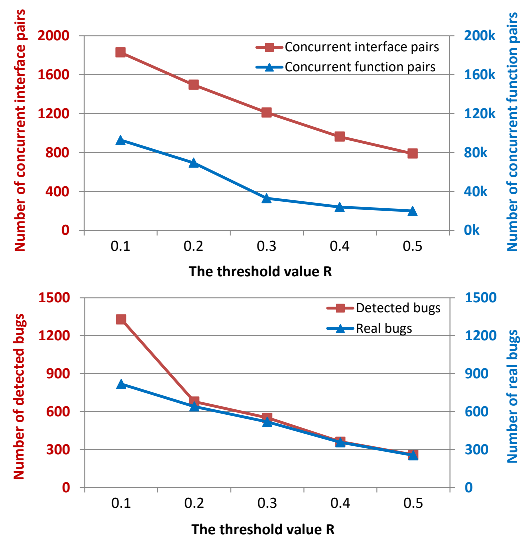


Figure 10: Variation of results by changing the value of R.

The numbers of extracted global concurrent interface pairs and extracted concurrent function pairs both decrease when R becomes larger. In this case, more false concurrent interface pairs and concurrent function pairs are dropped, but more of the dropped pairs are actually real concurrent interface pairs and concurrent function pairs, and thus the bugs involving these pairs are missed. Thus, if the value of R is too small, many false positives will be introduced, and if the value of R is too big, many false negatives will be introduced.

[1] https://lore.kernel.org/patchwork/patch/1025934/

5.4 False Positive and Negative Analysis

5.4.1 False Positives

DCUAF reports 33 and 39 false bugs in Linux 3.14 and 4.19, resulting in false positive rates of 5.9% and 5.7% respectively. These false bugs are introduced for the following reasons:

Firstly, the alias analysis in our lockset analysis is field-based. It cannot distinguish between different variables stored in the same data structure field, and thus it may identify different variables (for locks and data uses) as the same. This reason causes DCUAF to report 12 and 19 false bugs in Linux 3.14 and 4.19.

Secondly, our lockset analysis is flow-sensitive but does not validate path conditions. Thus, it may search infeasible code paths when detecting bugs. This reason causes DCUAF to report 8 and 5 false bugs in Linux 3.14 and 4.19.

Finally, our lockset analysis only handles lock-related function calls, but does not consider other kinds of synchronization. For example, the kernel interface `synchronize_irq` is used to wait until the end of an interrupt handler. Thus, code after the call to `synchronize_irq` should never be concurrently executed with an interrupt handler. But our lockset analysis does not consider this case. This reason causes DCUAF to report 13 and 15 false bugs in Linux 3.14 and 4.19.

Besides the reasons for the false positives observed in our evaluation, there are some other potential reasons for false positives that we have not yet observed in practice. For example, the value of R in our local-global strategy can largely influence the accuracy of extracting concurrent function pairs. If R is not properly set, some extracted global concurrent interface pairs and concurrent function pairs may be false. Moreover, unnecessary locks acquired in the driver code can also influence the accuracy of extracting concurrent function pairs. This case can occur when two driver functions should not be concurrently executed, but they acquire the same lock, because the driver developer is too conservative. Indeed, it has been observed that the Linux kernel does not provide systematic documentation about where locks should be used [28]. Unnecessary locks may cause DCUAF to identify the two driver functions as a concurrent function pair.

5.4.2 False Negatives

To analyze the false negatives of DCUAF, we compare its bug reports with the driver commits fixing concurrency use-after-free bugs identified in Section 2.3. Specifically, we focus on the commits in the few months after the release of Linux 4.19 in October 2018, *i.e.*, between October and December in 2018, resulting in 22 commits. DCUAF finds the bugs in 6 of these commits (including the commit in Figure 2), but misses the bugs in the remaining 16 commits. These bugs are missed for the following reasons:

Firstly, DCUAF lacks function pointer analysis in the local-global strategy and lockset analysis, and thus cannot build complete call graphs of the driver code. As a result, it cannot find real bugs involving code reached through function pointers. This reason causes DCUAF to miss the bugs in 2 commits.

Secondly, our alias analysis is field-based, and may err in complex cases, such as the cases involving function arguments and pointer assignments. It may identify two identical variables (for locks and data uses) as different variables. This reason causes DCUAF to miss the bugs in 4 commits.

Thirdly, our local-global strategy neglects some real cases of driver concurrency. For example, the strategy does not consider that a driver function can be concurrently executed with itself, or that driver functions can create new kernel threads. This reason causes DCUAF to miss the bugs in 2 commits.

Finally, DCUAF does not handle some other cases in driver code, which causes it to miss the bugs in 8 commits. For example, the RCU lock-acquiring functions do not have any argument, so DCUAF cannot use a lock argument to perform static analysis and find related bugs. Moreover, DCUAF does not consider the multi-queue framework that is used in some network and storage drivers. Besides, DCUAF does not consider reference count puts as possible freeing operations.

5.5 Sensitivity Analysis

DCUAF uses two key techniques: a local-global strategy to extract concurrent function pairs in driver code, and a summary-based lockset analysis to reduce repeated analysis. To better understand the value of these two techniques, we modify DCUAF to remove each of them, and evaluate each resulting tool on Linux 4.19 drivers.

Dropping the local-global strategy. We implement two tools by respectively following two assumptions used by previous approaches for detecting data races [37, 38]: (1) all driver interfaces can be concurrently executed; (2) driver interfaces whose field names containing some common keyword pairs for device initialization and deinitialization, including <probe, remove>, <start, stop>, <open, close>, <init, fini> and <resume, suspend>, cannot be concurrently executed . These amount to 257 pairs of driver interfaces. The first tool runs for 350 minutes and reports around 50K bugs. The second tool runs for 302 minutes and reports around 42K bugs. We found that most of the reported bugs found by these tools are false, because many involved driver interfaces that are never concurrently executed. Thus, our local-global strategy indeed reduces false positives in bug detection.

Dropping the summary-based lockset analysis. We implement this tool by dropping function summaries, keeping only the names of functions previously analyzed in the current execution path to avoid infinite loops due to recursion. The resulting tool runs for 850 minutes and then aborts due to insufficient memory. Thus, our summary-based analysis indeed improves the efficiency of the analysis.

6 Discussion

In this section, we discuss how our approach may apply to other kinds of driver problems.

Other concurrency bugs. DCUAF can be used to detect other kinds of concurrency bugs in drivers, by modifying the lockset analysis. For example, it can detect a data race in two driver functions that can be concurrently executed. We have implemented such a prototype approach based on DCUAF. It reports around 149K data races in Linux 4.19 drivers. However, many of the reported data races are benign. Thus, we have focused on a specific kind of serious concurrency bug, namely concurrency use-after-free bugs.

Violations of other properties of driver interfaces. In fact, which driver interfaces can be concurrently executed is an important property of driver interfaces. To identify this property, in DCUAF, we first collect specific code information in each driver and then perform a statistical analysis of the collected information. This idea can be used to identify other important properties of driver interfaces and detect related violations. An example property is whether a driver interface can sleep. If a driver interface is called in atomic context [10], this driver interface cannot call any function that can sleep. Otherwise, a sleep-in-atomic-context (SAC) bug will occur, which can cause a system hang or crash [4]. Following our local-global strategy in DCUAF, for a given driver interface, we can first collect the information about whether the related driver function calls sleep-able functions in each driver; then we can perform a statistical analysis of all the collected information to infer whether this driver interface is in atomic context; and finally using the inference results, we can detect SAC bugs.

7 Related Work

7.1 Detecting Use-After-Free Bugs

Many approaches [6,25,33,36,45] for detecting use-after-free bugs are based on dynamic analysis. They monitor memory accesses at runtime and report bugs according to exact runtime information. They can detect both sequential and concurrency use-after-free bugs. For example, DangSan [36] is an effective use-after-free detection system that can efficiently scale to large numbers of pointer writes and to many concurrent threads. To reduce the runtime overhead of monitoring pointer tracking, DangSan uses a lock-free design inspired by log-structured file systems. This design refrains from using complicated shared data structures and simply opts for append-only per-thread logs for each object in the common case. However, these approaches require workloads that can achieve good code coverage and bug-detection results, and they often introduce runtime overhead.

Several approaches [41, 42, 44] use static analysis to detect use-after-free bugs in user-mode applications. For example, UAFChecker [44] combines taint analysis and symbolic execution to find use-after-free bugs inter-procedurally. CRED [42] is an efficient pointer-analysis-based static analysis to detect use-after-free bugs in large code bases. It uses a spatio-temporal context reduction technique to reduce the exponential number of considered contexts in code analysis. It also uses a multi-stage analysis to efficiently filter out false alarms, and uses a path-sensitive demand-driven method to find the required points-to information.

These static approaches target use-after-free bugs that occur within sequential execution. To do this, they start dataflow analysis from a given free operation, and check whether there is a subsequent use operation. However, they do not consider bugs caused by concurrent execution. Different from these approaches, DCUAF targets concurrency use-after-free bugs. To do this, DCUAF starts alias analysis from two driver functions that may be concurrently executed. Besides, these approaches target user-mode applications, while DCUAF targets device drivers by considering the driver interface model.

7.2 Detecting Concurrency Problems

To detect concurrency problems in device drivers, many existing approaches are based on dynamic analysis or static analysis:

Dynamic analysis. Related dynamic analysis approaches are sampling-based [15, 18] or lockset-based [8, 20, 32]. DataCollider [15] is an effective sampling-based approach to detect data races in the Windows kernel. It randomly samples memory accesses at runtime. To increase the possibility of capturing concurrent accesses to identical memory addresses, it delays the current running thread for a short time, and uses hardware breakpoints to trap any second access during delay. If a second access happens and at least one is a write, a real data race is detected. Eraser [32] was the first lockset-based approach for detecting data races. It instruments binary code to perform runtime monitoring of shared-variable accesses for each running thread, and detects data races by maintaining and checking locksets of shared variables during execution.

Dynamic approaches require associated hardware devices to actually run the tested drivers, which may be hard to obtain in practice. Besides, due to the non-determinism of concurrent execution, they may miss many real concurrency bugs.

Static analysis. Most related static analysis approaches [12, 13, 17, 29, 37, 38] are based on static lockset analysis. RacerX [13] is a well-known static lockset-based approach for detecting data races and deadlocks in OS kernel code. It uses an inter-procedural, flow-sensitive and context-sensitive analysis to maintain and check locksets in code paths, and detects data races and deadlocks. It also ranks the reported bugs. WHOOP [12] is an efficient static lockset-based approach for detecting data races in device drivers. It uses a symbolic pairwise lockset analysis to attempt to prove a driver race-free. It also uses a sound partial-order reduction to accelerate CORRAL [21], an existing concurrency-bug detector.

These static approaches target general concurrency bugs such as data races and atomicity violations, and they often have many false positives (for example, the work on RacerX [13] reports a false positive rate of nearly 50%). They do not focus on concurrency use-after-free bugs. Besides, they assume that all driver interface functions can be concurrently executed [13, 37, 38] or rely on manual guidance [12, 17], which can introduce many false positives or require much manual work. Different from these approaches, DCUAF targets concurrency use-after-free bugs, and uses a local-global strategy to accurately and automatically extract concurrent function pairs from driver source code.

7.3 Mining Code Rules in Systems Software

Some approaches mine implicit code rules in systems software, and then use the mined rules to detect related bugs. They mine rules by statistically analyzing source code [14, 23, 26, 31, 46] or execution traces [5, 22, 43]. PR-Miner [26] uses data mining techniques to extract implicit programming rules from the source code of large code bases. It extracts frequent function-call patterns that occur within a single function. Using the extracted rules, it detects related violations in the source code. PairCheck [5] uses software fault injection to generate test cases that cover error handling code in device drivers, and then runs these test cases to mine resource-acquire and -release rules from execution traces. Using the mined rules, it detects resource leaks in error handling code.

These approaches focus on code rules that occur within sequential execution, such as resource-acquire and -release pairs [5, 31] and function-call sequences [26, 43], but do not consider code rules involving concurrency. Inspired by these approaches, DCUAF uses a statistical analysis of driver code information when extracting concurrent function pairs.

8 Conclusion

In this paper, we have proposed a practical static analysis approach named DCUAF, to effectively and automatically detect concurrency use-after-free bugs in Linux device drivers. DCUAF uses two key techniques: (1) a local-global strategy to extract the pairs of driver interface functions that may be concurrently executed as concurrent function pairs; (2) a summary-based lockset analysis to detect concurrency use-after-free bugs, given two driver functions that may be concurrently executed. We have evaluated DCUAF on the driver code of Linux 4.19, and found 640 real concurrency use-after-free bugs. We have randomly selected 130 of these real bugs and reported them to Linux kernel developers, and 95 have been confirmed.

DCUAF can be improved in some aspects. Firstly, the code analysis in DCUAF can be improved to reduce false positives. For example, DCUAF does not consider path conditions and non-lock-related synchronization primitives in its lockset analysis. Secondly, DCUAF still misses concurrency use-after-free bugs involving complex patterns, such as using reference counters to free objects. Runtime testing tools such as KASAN [19] have found some such bugs in Linux drivers. We will consider these complex patterns in our lockset analysis to find more concurrency use-after-free bugs. Thirdly, besides concurrency use-after-free bugs, DCUAF can be applied to other driver problems, including other concurrency bugs such as data races and violations of other properties of driver interfaces such as sleep-in-atomic-context bugs. Finally, DCUAF only checks Linux drivers at present. We will port DCUAF in other operating systems (such as FreeBSD and NetBSD) to check their driver code.

Acknowledgment

We thank our shepherd Nadav Amit and the anonymous reviewers for their helpful advice on the paper. We also thank the Linux kernel developers who gave useful feedback to us.

References

[1] BAI, J.-J. Linux kernel commit 2ff33d663739: fix some concurrency double-free bugs in the isdn_tty driver. https://github.com/torvalds/linux/commit/2ff33d663739.

[2] BAI, J.-J. Linux kernel commit 4f68ef64cd7f: fix some concurrency use-after-free bugs in the cw1200 driver. https://github.com/torvalds/linux/commit/4f68ef64cd7f.

[3] BAI, J.-J. Linux kernel commit 7418e6520f22: fix a concurrency use-after-free bug in the hfc_pci driver. https://github.com/torvalds/linux/commit/7418e6520f22.

[4] BAI, J.-J., WANG, Y.-P., LAWALL, J., AND HU, S.-M. DSAC: effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference* (2018), pp. 587–600.

[5] BAI, J.-J., WANG, Y.-P., LIU, H.-Q., AND HU, S.-M. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology 73* (2016), 122–133.

[6] CABALLERO, J., GRIECO, G., MARRON, M., AND NAPPA, A. 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), pp. 133–143.

[7] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel

vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)* (2011), pp. 1–5.

[8] CHEN, Q.-L., BAI, J.-J., JIANG, Z.-M., LAWALL, J., AND HU, S.-M. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 366–376.

[9] Clang compiler. http://clang.llvm.org/.

[10] CORBET, J. Atomic context and kernel api design, 2008. https://lwn.net/Articles/274695/.

[11] Coverity. https://scan.coverity.com.

[12] DELIGIANNIS, P., DONALDSON, A. F., AND RAKAMARIC, Z. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)* (2015), pp. 166–177.

[13] ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)* (2003), pp. 237–252.

[14] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP)* (2001), pp. 57–72.

[15] ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., AND OLYNYK, K. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)* (2010), pp. 151–162.

[16] HEINTZE, N., AND TARDIEU, O. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 2001 International Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 254–263.

[17] HONG, S., AND KIM, M. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software (JSS) 86*, 2 (2013), 377–388.

[18] JIANG, Y., YANG, Y., XIAO, T., SHENG, T., AND CHEN, W. DRDDR: a lightweight method to detect data races in Linux kernel. *The Journal of Supercomputing 72*, 4 (2016), 1645–1659.

[19] The Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[20] KernelStrider: Detecting data races in Linux kernel modules by collecting runtime information. https://github.com/euspectre/kernel-strider.

[21] LAL, A., QADEER, S., AND LAHIRI, S. K. A solver for reachability modulo theories. In *Proceedings of the 2012 International Conference on Computer Aided Verification (CAV)* (2012), pp. 427–443.

[22] LAROSA, C., XIONG, L., AND MANDELBERG, K. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing* (2008), pp. 880–885.

[23] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 43–52.

[24] Linux Driver Verification. http://linuxtesting.org/ldv.

[25] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)* (2015).

[26] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering (FSE)* (2005), pp. 306–315.

[27] Linux kernel source tree. https://github.com/torvalds/linux.

[28] LOCHMANN, A., SCHIRMEIER, H., BORGHORST, H., AND SPINCZYK, O. LockDoc: trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)* (2019), pp. 11:1–11:15.

[29] LU, S., PARK, S., HU, C., MA, X., JIANG, W., LI, Z., POPA, R. A., AND ZHOU, Y. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st International Symposium on Operating Systems Principles (SOSP)* (2007), pp. 103–116.

[30] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)* (2008), pp. 247–260.

[31] SAHA, S., LOZI, J.-P., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[32] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 391–411.

[33] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), pp. 309–318.

[34] Syzkaller: an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[35] The USB related documentations in the Linux kernel. https://www.kernel.org/doc/Documentation/usb/.

[36] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (2017), pp. 405–419.

[37] VOJDANI, V., APINIS, K., RÕTOV, V., SEIDL, H., VENE, V., AND VOGLER, R. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)* (2016), pp. 391–402.

[38] VOUNG, J. W., JHALA, R., AND LERNER, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the 2007 International Symposium on Foundations of Software Engineering (FSE)* (2007), pp. 205–214.

[39] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. AsyncShock: exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of the 2016 European Symposium on Research in Computer Security (ESORICS)* (2016), pp. 440–457.

[40] XU, W., LI, J., SHU, J., YANG, W., XIE, T., ZHANG, Y., AND GU, D. From collision to exploitation: unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)* (2015), pp. 414–425.

[41] YAN, H., SUI, Y., CHEN, S., AND XUE, J. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)* (2017), pp. 42–54.

[42] YAN, H., SUI, Y., CHEN, S., AND XUE, J. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (2018), pp. 327–337.

[43] YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE)* (2006), pp. 282–291.

[44] YE, J., ZHANG, C., AND HAN, X. UAFChecker: scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 21st International Conference on Computer and Communications Security (CCS)* (2014), pp. 1529–1531.

[45] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)* (2015).

[46] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 2016 USENIX Security Symposium* (2016), pp. 363–378.

LXDs: Towards Isolation of Kernel Subsystems

Vikram Narayanan
University of California, Irvine

Abhiram Balasubramanian*
University of Utah

Charlie Jacobsen*
University of Utah

Sarah Spall†
University of Utah

Scott Bauer‡
University of Utah

Michael Quigley§
University of Utah

Aftab Hussain
University of California, Irvine

Abdullah Younis¶
University of California, Irvine

Junjie Shen
University of California, Irvine

Moinak Bhattacharyya
University of California, Irvine

Anton Burtsev
University of California, Irvine

Abstract

Modern operating systems are monolithic. Today, however, lack of isolation is one of the main factors undermining security of the kernel. Inherent complexity of the kernel code and rapid development pace combined with the use of unsafe, low-level programming language results in a steady stream of errors. Even after decades of efforts to make commodity kernels more secure, i.e., development of numerous static and dynamic approaches aimed to prevent exploitation of most common errors, several hundreds of serious kernel vulnerabilities are reported every year. Unfortunately, in a monolithic kernel a single exploitable vulnerability potentially provides an attacker with access to the entire kernel.

Modern kernels need isolation as a practical means of confining the effects of exploits to individual kernel subsystems. Historically, introducing isolation in the kernel is hard. First, commodity hardware interfaces provide no support for efficient, fine-grained isolation. Second, the complexity of a modern kernel prevents a naive decomposition effort. Our work on Lightweight Execution Domains (LXDs) takes a step towards enabling isolation in a full-featured operating system kernel. LXDs allow one to take an existing kernel subsystem and run it inside an isolated domain with minimal or no modifications and with a minimal overhead. We evaluate our approach by developing isolated versions of several performance-critical device drivers in the Linux kernel.

1 Introduction

Modern operating system kernels are fundamentally insecure. Due to rapid development rate (the de-facto industry standard Linux kernel features over 70 thousand commits a year), a huge codebase (the latest version of the Linux kernel contains over 17 million lines of unsafe C/C++ and assembly code[1]), and inherent complexity (typical kernel code adheres to multiple allocation, synchronization, access control, and object lifetime conventions) bugs and vulnerabilities are routinely introduced into the kernel code. In 2018, the Common Vulnerabilities and Exposures database lists 176 Linux kernel vulnerabilities that allow for privilege escalation, denial-of-service, and other exploits [20]. This number is the lowest across several years [19].

Even though a number of static and dynamic mechanisms have been invented to protect execution of the low-level kernel code, e.g., modern kernels deploy stack guards [18], address space layout randomization (ASLR) [45], and data execution prevention (DEP) [72], attackers come up with new ways to bypass these protection mechanisms [8, 35, 45, 49, 51, 57–59, 71]. Even advanced defense mechanisms that are yet to be deployed in mainstream kernels, e.g., code pointer integrity (CPI) [1, 53] and safe stacks [14, 53], become vulnerable in the face of data-only attacks combined with automated attack generation tools [46, 77]. In a monolithic kernel, a single vulnerability provides an attacker with access to the entire kernel. An attacker can redirect control to any part of the kernel and change any data structure escalating its privileges [46, 77].

Modern kernels need isolation as a practical means of confining the effects of individual vulnerabilities. However, introducing isolation in a kernel is hard. First, despite many advances in the architecture of modern CPUs, low-overhead hardware isolation mechanisms [74–76] did not make it into commodity architectures. On modern machines, the minimal call/reply invocation that relies on traditional address-spaces for isolation [26] takes over 834 cycles (Section 5). To put this number into perspective, in the Linux kernel a system call that sends a network packet through the network stack and network device driver takes 2299 cycles. A straightforward isolation of a network device driver which requires two domain crossings on the packet transmission path (Section 4), would introduce an overhead of more than 72%.

Second, the complexity of a shared-memory kernel that accumulates decades of development in a monolithic setting

*Currently at Ubiquiti Networks. Work done at the University of Utah.
†Currently at Indiana University. Work done at the University of Utah.
‡Currently at Qualcomm. Work done at the University of Utah.
§Currently at Google. Work done at the University of Utah.
¶Currently at the University of California, Berkeley. Work done at the University of California, Irvine
[1]Calculated using David Wheeler's sloccount on Linux 5.0-rc1.

prevents a trivial decomposition effort. Decomposition requires cutting through a number of tightly-connected, well-optimized subsystems that use rich interfaces and complex interaction patterns. Two straightforward isolation strategies—developing isolated subsystems from scratch [21, 24, 31] or running them inside a full copy of a virtualized kernel [11, 16, 30, 60]—result in either a prohibitively large engineering effort or overheads of running a full copy of a kernel for each isolated domain.

Our work on Lightweight Execution Domains (LXDs) takes a step towards enabling isolation in a full-featured operating system kernel. LXDs allow one to take an existing kernel subsystem and run it inside an isolated domain with minimal or no modifications and with a minimal overhead. While isolation of core kernel subsystems, e.g., a buffer cache layer, is beyond the scope of our work due to tight integration with the kernel (i.e., complex isolation boundary and frequent domain crossings), practical isolation of device drivers, which account for over 11 millions lines of unsafe kernel code and significant fraction of kernel exploits, is feasible.

Compared to prior isolation attempts [9, 12, 15, 27, 32–34, 40–43, 66, 68, 69], LXDs leverage several new design decisions. First, we make an observation that synchronous cross-domain invocations are prohibitively expensive. The only way to make isolation practical is to leverage asynchronous communication mechanisms that batch and pipeline multiple cross-domain invocations. Unfortunately, explicit management of asynchronous messages typically requires a clean-slate kernel implementation built for explicit message-passing [5, 42]. LXDs, however, aim to enable isolation in commodity OS kernels that are originally monolithic (commodity kernels accumulate decades of software engineering effort that is worth preserving). To introduce asynchronous communication primitives in the code of a legacy kernel, LXDs build on the ideas from asynchronous programming languages [3,13,39]. We develop a lightweight asynchronous runtime that allows us to create lightweight cooperative threads that may block on cross-domain invocations and hence implement batching and pipelining of cross-domain calls in a way transparent to the kernel code.

Second, to break the kernel apart in a manner that requires only minimal changes to the kernel code, we develop *decomposition patterns*, a collection of principles and mechanisms that allow decomposition of the monolithic kernel code. Specifically, we support decomposition of typical idioms used in the kernel code—exported functions, data structures passed by reference, function pointers, etc. To achieve such backward compatibility, decomposition patterns define a minimal runtime layer that hides isolated, share-nothing environment by synchronizing private copies of data structures, invoking functions across domain boundaries, implementing exchange of pointers to data structures and functions, handling dispatch of cross-domain function calls, etc. Further, to make our approach practical, we develop an interface definition language (IDL) that generates runtime glue-code code required for decomposition.

Finally, similar to existing projects [54, 67], we make an observation that on modern hardware cross-core communication via the cache coherence protocol is faster then crossing an isolation boundary on the same CPU. By placing isolated subsystems on different cores it is possible to reduce isolation costs. While dedicating cores for every isolated driver is impractical, the ability to run several performance-critical subsystems, e.g., NVMe block and network device drivers, with the lowest possible overhead makes sense.

We demonstrate practical isolation of several performance-critical device drivers in the Linux kernel: software-only network and NVMe block drivers, and a 10Gbps Intel ixgbe network driver. Our experience with decomposition patterns shows that majority of the decomposition effort can be done with no modification to the kernel source. We hope that our work—general decomposition patterns, interface definition language, and asynchronous execution primitives—will gradually enable kernels that employ fine-grained isolation as the first-class abstraction. At the moment, two main limitations of LXDs are 1) requirement of a dedicated core for each thread of an isolated driver (Section 5), and 2) manual development of the IDL interfaces. We expect to relax both of the limitations in our future work.

2 Background and Motivation

The concept of decomposing operating systems for isolation and security is not new [9,12,15,27,32–34,40–43,66,68,69]. In the past, multiple projects tried to make isolation practical in both microkernel [9, 27, 41–43, 68] and virtual machine [12, 15, 33, 66] systems. SawMill was a research effort performed by IBM aimed at building a decomposed Linux environment on top of the L4 microkernel [34]. SawMill was an ambitious effort that outlined many problems of fine-grained isolation in OS kernels. SawMill relied on a synchronous IPC mechanism and a simple execution model in which threads migrated between isolated domains. Unfortunately, the cost of a synchronous context switch more than doubled in terms of CPU cycles over the last two decades [26]. Arguably, with existing hardware mechanisms the choice of a synchronous IPC is not practical (on our hardware a bare-bone synchronous call/reply invocation takes over 834 cycles on a 2.6GHz Intel machine; a cache-coherent invocation between two cores of the same die takes only 448-533 cycles, moreover, this number can be reduced further with batching (Section 5)). Furthermore, relying on a generic Flick IDL [25], SawMill required re-implementation of OS subsystem interfaces. In contrast, LXDs's IDL is designed with an explicit goal of backward compatibility with the existing monolithic code, i.e., we develop mechanisms that allow us to transparently support decomposition of typical code patterns used in the kernel, e.g., registration of interfaces as function pointers, passing data structures by reference, etc.

Nooks further explored the idea of isolating device drivers in the Linux kernel [69]. Similar to SawMill, Nooks relied on the synchronous cross-domain procedure calls that are prohibitively expensive on modern hardware. Nooks maintained and synchronized private copies of kernel objects, however, the synchronization code had to be developed manually. Nooks' successors, Decaf [64] and Microdrivers [32] developed static analysis techniques to generate glue code directly from the kernel source. LXDs do not have a static analysis support at the moment. We, however, argue that IDL is still an important part of a decomposed architecture—IDL provides a generic intermediate representation that allows us to generate glue code for different isolation boundaries, e.g., cross-core invocations, address-space switches, etc.

OSKit developed a set of decomposed kernel subsystems out of which a full-featured OS kernel could be constructed [29]. While successful, OSKit was not a sustainable effort—decomposition glue code was developed manually, and required a massive engineering effort in order to provide compatibility with the interface of Component Object Model [17]. OSKit quickly became outdated and unsupported.

Rump kernels develop glue code that allows execution of unmodified subsystems of the NetBSD kernel in a variety of executable configurations on top of a minimal execution environment [48], e.g., as a library operating system re-composed out of Rump kernel subsystems. Rump's glue code follows the shape of the kernel subsystems and hence provides compatibility with unmodified kernel code that ensures maintainability of the project. LXDs follow Rump's design choice of ensuring backward compatibility with unmodified code, but aim at automating the decomposition effort. Specifically, LXDs rely on decomposition patterns and IDL to extract unmodified device drivers from the kernel source and seamlessly enable their functionality for the monolithic kernel.

User-level device drivers [11, 21, 24, 30, 60] allow execution of device drivers in isolation. Two general approaches are used for isolating the driver. First, it is possible to run an unmodified device driver on top of a device driver execution environment that provides a backward compatible interface of the kernel inside an isolated domain [24]. Unfortunately, development of a kernel-compatible device driver execution environment requires a large engineering effort. Sometimes, backward compatibility is sacrificed to simplify development, but in this case the device driver or a kernel subsystem have to be re-implemented from scratch [31]. LXDs aim to provide a general framework for automating development of custom backward compatible device driver environments. With a powerful IDL, fast communication primitives, and asynchronous threads, LXDs enable nearly transparent decomposition of kernel code.

Alternatively, the device driver environment is constructed from a partial or complete copy of the kernel that can host the isolated driver on top of a VMM [11, 30, 60] or inside a user process [11, 48]. Unfortunately, a virtualized kernel [11, 30, 60] extends the driver execution environment with a nested copy of multiple software layers, e.g., interrupt handling, thread scheduling, context-switching, memory management, etc. These layers introduce overheads of tens of thousands of cycles on the critical data-path of the isolated driver, and provide a large attack surface. A library operating system that provides full or partial compatibility with the original kernel can be used as an execution environment for the isolated device driver [48, 62, 70]. Smaller and lighter compared to the full kernel, library operating systems eliminate performance overheads of the full kernel. LXDs provide ability to run an unmodified device driver in a very minimal kernel environment hence achieving lean data path of a custom-built device driver execution environment.

3 LXDs Architecture

LXDs execute as a collection of isolated domains running side by side with the monolithic kernel (Figure 1). This design allows us to enable isolation incrementally, i.e., develop isolated device drivers one at a time, and seamlessly enable their functionality in the monolithic kernel.

Each LXD is developed as a loadable kernel module. An unmodified source of the isolated driver is linked against the two components that provide a backward compatible execution environment for the driver: 1) the glue code generated by the IDL compiler Figure 1, ⑥), and 2) a minimal library, libLXD (Figure 1, ⑦), that provides common utility functions normally available to the driver in a monolithic kernel, e.g., memory allocators, synchronization primitives, routines like memcpy(), etc.

LXDs rely on hardware-assisted virtualization (VT-x) for isolation. The choice of the hardware isolation mechanism is orthogonal to the LXDs architecture. VT-x, however, implements convenient interface for direct assignment of PCIe devices to isolated domains, and direct interrupt delivery (support for which we envision in the future). On the critical path LXDs rely on asynchronous cross-core communication primitives, and hence the cost of transitions to and from the VT-x domain (which is higher than a regular context switch) is acceptable.

LXDs are created and managed by a small microkernel that runs inside the commodity operating system kernel (Figure 1, ⑧). The LXD microkernel follows design of the L4 microkernel family [26]: it is centered around a pure capability-based synchronous IPC that explicitly controls authority of each isolated subsystem. The synchronous IPC is used for requesting microkernel resources, and exchange of capabilities, e.g., establishing regions of shared memory that are then used for fast asynchronous channels. Each LXD starts with at least one synchronous IPC channel that allows the LXD to gain more capabilities, exchange capabilities to its memory pages with the non-isolated kernel, and establish fast asynchronous communication channels.

To provide an interface of the isolated driver inside the

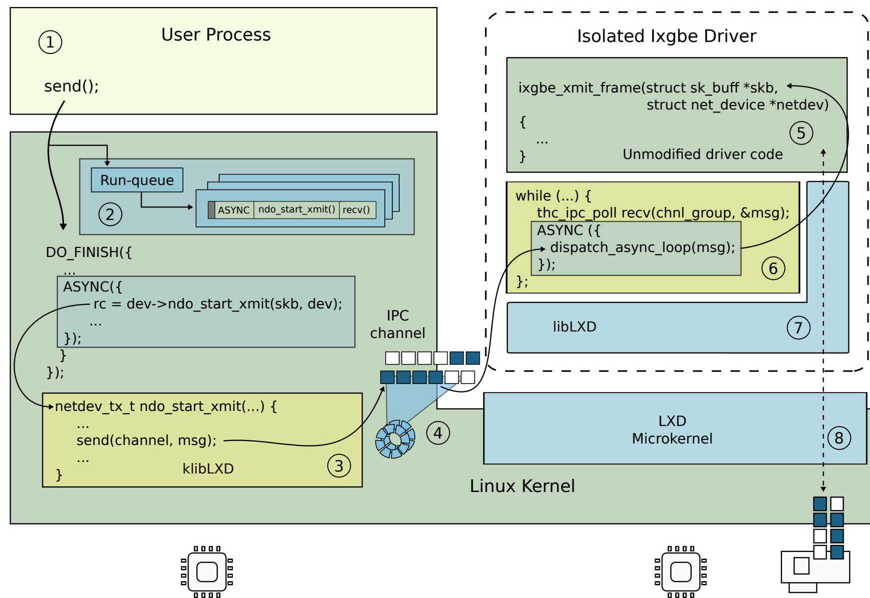


Figure 1: LXDs architecture (isolated ixgbe network driver).

monolithic kernel, every LXD loads the corresponding klibLXD module (Figure 1, ③). The klibLXD is automatically generated by the IDL compiler. The glue code inside klibLXD transparently marshals arguments of cross-domain invocations to the actual isolated driver.

In the example of the isolated ixgbe driver (Figure 1), a user process invokes an unmodified send() system call initiating transmission of the network packet through the isolated device driver (Figure 1, ①). The monolithic kernel relies on the interface of the isolated ixgbe driver (a collection of function pointers registered by the driver with the kernel) to pass the packet to the device (dev−>ndo_start_xmit()). The dev−>ndo_start_xmit() function pointer is implemented by the glue code of the klibLXD module. Internally, the glue code relies on the low-level send and receive primitives of the asynchronous communication channels to send the message to the isolated driver. The message reaches the isolated driver where it is processed by the dispatch loop generated by the IDL compiler (Figure 1, ⑥). The dispatch loop then invokes the actual ixgbe_xmit_frame() function of the unmodified ixgbe driver (Figure 1, ⑤).

Transparent decomposition LXDs rely on a collection of decomposition patterns to break the code of a monolithic system and emulate a shared-memory environment for isolated subsystems (Section 4). In LXDs, isolated subsystems do not share any state that might break isolation guarantees, e.g., memory pointers, indexes into memory buffers, etc. Instead, each isolated subsystem maintains its own private hierarchy of data structures. LXDs rely on a powerful IDL to automatically generate all inter-domain communication and synchronization code (Figure 1, ③ and ⑥). In contrast to existing IDLs used for constructing multi-server [25, 38] and distributed systems [23, 50, 61, 73] the main design goal behind the LXDs' IDL is backward compatibility with unmodified code. The IDL is designed to generate caller and callee stubs that hide details of inter-domain communication and synchronization of data structures.

Asynchronous runtime Compared to the monolithic kernel, a decomposed environment requires a cross-domain invocation in place of a regular procedure call for every function that crosses the boundary of an isolated domain. On modern hardware the overhead of such crossings is prohibitively expensive. LXDs include a minimal runtime built around lightweight asynchronous threads that aims to hide overheads of cross-domain invocations by exploiting available request parallelism. Specifically, the ASYNC() primitive creates a lightweight cooperative thread that yields execution to the next thread when blocked on the reply from an isolated domain (Figure 1, ②). Asynchronous threads allow us to introduce asynchrony to the kernel code in a transparent manner.

Cross-core IPC To reduce overheads of crossing domain boundaries, LXDs schedule isolated subsystems with tight latency and throughput requirements, i.e., network and block device drivers, on separate CPU cores. The reason is that on modern hardware cross-core communication via cache coherence is faster than a context switch on the same CPU. LXDs rely on efficient cross-core communication channels to send messages across isolated subsystems (Figure 1, ④).

3.1 Interface Definition Language

We develop a collection of *decomposition patterns*— a collection of principles and mechanisms, e.g., remote references, projections, and hidden arguments, that allow isolation of typical code patterns used in the kernel, e.g. exported functions, data structures passed by reference, registration of interfaces

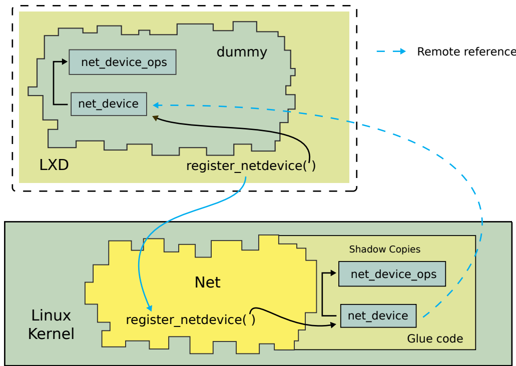


Figure 2: Private object hierarchies.

as function pointers, etc. To support implementation of decomposition patterns, we develop a powerful IDL that generates all inter-domain communication code.

Modules The IDL describes each subsystem as a *module*, i.e., a collection of functions exported and imported by an isolated driver or the kernel. To illustrate decomposition patterns and the design of the IDL, we consider an example of a minimal dummy network device driver [44]. The following IDL is used to define the dummy module.

```
include <net.idl>
module dummy() {
    require net;
}
```

By itself the dummy module does not export any functions. Instead it relies on the net interface provided by the kernel to register itself with the kernel, i.e., register a collection of function pointers that provide the driver-specific implementation of the network device interface. The kernel uses these function pointers to invoke the isolated dummy device driver.

The require keyword instructs the IDL compiler to import the net module into the context of the dummy module. At a high level, the compiler is instructed to generate glue code required for remote invocations of the functions exported by the net module.

A typical network interface defines a collection of functions that implement a specific kernel interface. For example, the net module defines the interface of the network subsystem, i.e., a collection of functions that allow network device drivers to register with the kernel.

```
module net() {
  rpc int register_netdevice(projection net_device *dev);
  rpc void ether_setup(projection net_device *dev);
  ...
}
```

From the above module definition the IDL generates code for caller stubs of the net interface so the isolated dummy module can transparently invoke functions of the interface. The IDL also generates the dispatch loops for both the dummy LXD and kLXD so both isolated subsystem and non-isolated kernel can process remote function invocations from each other.

Data structures In LXDs, isolated device drivers and the kernel do not share any state that might break isolation guarantees. Instead, each isolated subsystem maintains its own private hierarchy of data structures. In our example, the register_netdev() function takes a pointer to the net_device data structure that describes the network device. Since net_device is allocated inside the isolated dummy driver, a corresponding shadow copy will be created by the glue code in the non-isolated kernel (Figure 2).

The shadow hierarchies are synchronized upon function invocations. LXDs provide support for transparent synchronization of shadow data structure copies across domains with the mechanism of *projections*. A projection explicitly defines a subset of fields of the data structure that will be passed to the callee and returned to the caller during the domain invocation.[2]

```
projection <struct net_device> net_device {
  unsigned int flags;
  unsigned int priv_flags;
  ...
  projection net_device_ops [alloc(caller)] *netdev_ops;
}
```

Here, the projection net_device only lists the fields that will be used by the non-decomposed code in the kernel to register a network device. The projection omits the members of struct net_device that are private to the LXD, e.g., pointers to other data structures. The IDL supports lexical scopes, so the same data structure can be projected differently by different functions.

The IDL supports explicit [in] and [out] directional attributes to specify whether each field is marshalled from caller to callee or vice versa. In most cases, however, they are optional. The IDL compiler can infer the default direction from the way the projection is used in the code. In the example above, the default direction is [in]—all fields of the projection are copied from the caller to the callee side, which is decided based on the [alloc(callee)] qualifier that we discuss below.

Allocation of shadow object copies When the register_netdev() function is invoked by the LXD, the callee side of the invocation, i.e., the non-decomposed kernel, does not have a private version of the net_device data structure. The IDL provides support for controlling when remote objects are allocated, looked up, and freed with the alloc, bind, and dealloc qualifiers. The alloc qualifier instructs the IDL to allocate the new data structure of the projected type, i.e., struct net_device. The callee attribute instructs the IDL to perform the allocation on the callee side, as the data structure already exists on the caller side. The allocation attribute also serves as a hint to the compiler to marshal all fields of the projection from the already existing data

[2]Hence defining how a data structure is projected into another domain.

structure on the caller side to the callee (i.e., all fields of the projection above have the implicit [in] attribute). The data structure is deallocated when the dealloc qualifier is used with the projection.

Remote object references In most cases isolated subsystems refer to the same data structure multiple times. For example, the net_device data structure is first registered with the register_netdev() function, then used in a number of functions that attach, turn on, and eventually unregister the device. LXDs provide a mechanism of *remote references* to refer to a specific object across domain boundaries. Similar to a capability in the LXD microkernel, each remote reference is a number that is resolved through a fast hash that is private to each thread of execution. References are transparent to the code, the IDL generates all necessary code to pair every local object with a reference that is used to lookup a corresponding shadow copy in another domain.

Function pointers Many parts of the kernel rely on the concept of an interface that allows dynamic registration of a specific subsystem implementation. In a native language like C an interface is implemented as a data structure with a collection of function pointers that are defined by each subsystem that provides a concrete interface implementation. In our example, net_device_ops is a data structure that defines a collection of function pointers implemented by the network device. We implement support for export of function pointers that cross boundaries of isolated domains. The following code provides a definition of the projection for the net_device_ops data structure.

```
projection <struct net_device_ops> net_device_ops {
  rpc [alloc] int (*ndo_open)(projection netdev_empty [bind] *dev);
  rpc [alloc] int (*ndo_stop)(projection netdev_empty [bind] *dev);
  rpc [alloc] int (*ndo_start_xmit)(projection sk_buff *skb,
                        projection net_device [bind] *dev);
  ...
}
```

For every function pointer, the IDL generates caller and callee stubs that behave like normal function pointers and hide details of cross-domain communication. To implement cross-domain function pointers while providing unmodified function signatures, we implement a concept of *hidden arguments*. For each function pointer, the IDL dynamically generates an executable trampoline in the caller's address space. The caller invokes this trampoline like any other function, however, the trampoline resolves additional hidden arguments as an offset from its own address. The hidden arguments describe which channel to use and passes this information to the cross-domain stub generated by the IDL compiler. A remote reference to a function pointer on the callee side allows the caller to resolve a specific instance of a function pointer.

Implementation We implement the IDL compiler as a source-to-source translator from the LXD IDL to C. To build the compiler, we rely on the formalism of parsing expression grammars (PEG). This choice allows us to design a modular grammar that is easy to extend with new IDL primitives. We use Vembyr PEG parser generator [63] to automate development of a compiler. Vembyr provides a convenient extension interface that allows us to construct an abstract syntax tree (AST) as a set of C++ classes. We then perform a compilation step as a series of passes over the AST, e.g., module import, derivation of directional attributes, etc.. The final pass converts the AST into a concrete syntax tree (CST) that we use to print out the C code.

```
1  DO_FINISH({
2    while (skb) {
3      struct sk_buff *next = skb->next;
4      ASYNC({
5        ...
6        rc = ndo_start_xmit(skb, dev);
7        ...
8      });
9      skb = next;
10   }
11 });
```

Listing 1: Asynchronous threads.

3.2 Asynchronous Execution Runtime

Traditionally, asynchronous communication requires explicit message passing [5, 42]. Programming of asynchronous message-passing systems, however, is challenging as it requires manual management (saving and restoring) of computation as execution gets blocked on remote invocations. In general, message-based systems work well as long as they are limited to a simple run-to-completion loop, but become nearly impossible to program if multiple blocking invocations are required on the message processing path [2, 52]. Further, in a message-passing environment, re-use of existing kernel code becomes hard or even impossible.

With LXDs we aim to satisfy two contradicting goals: 1) utilize asynchrony for cross-domain invocations, and 2) provide backward compatibility with existing kernel code, i.e., avoid re-implementation of the system in a message-passing style. To meet these goals, we implement a lightweight runtime that hides details of asynchronous communication behind an interface of asynchronous threads.

The core of the LXDs asynchronous runtime is built around two primitives: ASYNC() and DO_FINISH(). In Listing 1 the ASYNC() macro creates a new lightweight thread for executing a block of code (lines 4–8) asynchronously. Our implementation is based on GCC macros as it allows us to avoid modifications to the compiler and therefore provides compatibility with the existing kernel toolchain. When ndo_start_xmit() blocks on sending a message to the isolated driver (line 6), the asynchronous runtime continues execution from the next line after the asynchronous block (line 9) and starts the next iteration of the loop creating the new asynchronous thread.

Instead of blocking on the first ndo_start_xmit() cross-domain invocation, we dispatch multiple asynchronous invocations, hence submitting multiple network packets to the driver in a pipelined manner.

Internally, ASYNC() creates a minimal thread of execution by allocating a new stack and switching to it for execution of the code inside of the asynchronous block. ASYNC() creates a continuation, i.e., it saves the point of execution that follows the asynchronous block, which allows the runtime to resume execution when the thread either blocks or finishes. We save the state of the thread, i.e., its callee saved registers, on the stack, and therefore, can represent continuation as a tuple {instruction pointer, stack pointer}. The continuation is added to the run-queue that holds all asynchronous threads that are created in the context of the current kernel thread. When asynchronous thread blocks waiting on a reply from an isolated domain, it invokes the yield() function that again saves the state of the thread by creating another continuation that is added to the run-queue. The yield() function picks the next continuation from the run-queue and switches to it.

The DO_FINISH() macro specifies the scope in which all asynchronous threads must complete. When execution reaches the end of the DO_FINISH() block, the runtime checks if any of unfinished asynchronous threads are still on the run-queue. If yes, the runtime creates a continuation for the current thread knowing that it has to finish the DO_FINISH() block later, and switches to a thread from the run-queue.

Integration with the messaging system Every time a remote invocation blocks waiting on a reply, the asynchronous runtime switches to the new thread. The runtime system checks the reply message ring for incoming messages and whether any of them can unblock one of the blocked asynchronous threads. We implement a lightweight data structure that allows us to resolve response identifiers into pointers to asynchronous threads waiting on the run-queue. If the response channel is empty, the runtime system tries to return to the main thread to dispatch more asynchronous threads, but if the main thread reached the end of the DO_FINISH() block it picks one of the existing threads from the run-queue.

Nested invocations In most cases a cross-domain invocation triggers one or more nested remote invocations back into the caller domain, be it an LXD or a non-isolated kernel. For example, the ndo_start_xmit() triggers invocation of the consume_skb() function that releases the skb after it is sent. We need to process nested invocations in the caller domain. To avoid using an extra thread to dispatch remote invocations, we process nested invocations in the context of the caller thread. We embed a dispatch loop, the optimization we call "sender's dispatch loop", inside the message receive function thc_ipc_poll_recv() in such a way that it listens and processes incoming invocations from the callee.

Implementation ASYNC() and DO_FINISH() leverage functionality of GCC macros that allow us to declare the block of code as a nested function that can be executed on a new stack. We base implementation of the threading runtime on the eager version of AC [39] (in LXDs cross-domain invocations always block, therefore, eager creation of the stack for each asynchronous thread is justified). Besides changing AC to work inside the Linux kernel and integrating it with the LXDs messaging primitives, we employ several aggressive optimizations. To minimize the number of thread switches, we introduce an idea of a "direct" continuation, the continuation that is known to follow the current context of execution, e.g., the instruction following the ASYNC() block. We also defer deallocation of stacks. Normally, the stack cannot be deallocated from the context of the thread that is using it. AC switches into the context of the "idle" scheduler thread and deallocates the stack from there. This, however, introduces an extra context switch. Instead, we maintain the queue of stacks pending de-allocation and deallocate them all at once right when the execution exits the DO_FINISH() block.

3.3 Fast Cross-Core Messaging

Trying to reduce overheads of crossing the isolation boundary, LXDs schedule isolated subsystems on separate CPU cores and use a fast cross-core communication mechanism to send "call" and "reply" invocations between the cores. The performance of cross-core invocations is dominated by the latency of cache coherence protocol, which synchronizes cache lines between cores (a single cache-line transaction incurs a latency of 100-400 cycles [22, 56, 57]). In order to achieve the lowest possible communication overhead, similar to prior projects [5,6,47], we minimize the number of cache coherence transactions. In LXDs, each channel consists of two rings: one for outgoing "call" messages and one for incoming "replies". We configure each message to be the size of a single cache line (64 bytes on our hardware). Similar to FastForward [36], we avoid shared producer and consumer pointers, as they add extra transactions for each message. Instead, we utilize an explicit state flag that signals whether the ring slot is free.

4 Decomposition Case-Studies

To evaluate the generality of LXDs abstractions, we develop several isolated device drivers in the Linux kernel.

4.1 Network Device Drivers

We develop isolated versions of two network drivers: 1) a software-only dummy network driver that emulates an infinitely fast network adapter, and 2) Intel 82599 10Gbps Ethernet driver (ixgbe). The network layer of the Linux kernel has one of the tightest performance budgets among all kernel subsystems. Further, the dummy is not connected to a real network interface, and hence allows us to stress overheads of isolation without any artificial limits of existing NICs.

Decomposed network drivers To isolate the dummy and ixgbe network drivers, we develop IDL specifications of the network driver interface. The IDL specification is 64 lines of

code for the dummy, and 153 lines for the ixgbe driver (110 lines for the network and 43 for the PCIe bus interfaces). Each network device driver registers with the kernel by invoking a kernel function and passing a collection of function pointers that implement an interface of a specific driver. Since ixgbe manages a real PCIe device, it registers with the PCIe bus driver that enumerates all PCIe devices on the bus and connects them to matching device drivers. Therefore, we develop an IDL specification for the PCI bus interface.

In contrast to block device drivers that implement a zero-copy path for block requests, the network stack copies each packet from a user process into a freshly allocated kernel buffer. To ensure a zero-copy transfer of the packet from the kernel to the LXD, we allocate a region of memory shared between the non-isolated kernel and the LXD of the network driver. The kernel allocates memory for the skb payload using the alloc_skb() function. We modify it to allocate payload data from this shared region. Linux does not provide a simple mechanism to configure one of its memory allocators to run on a specified region of memory. We, therefore, develop a lock-free allocator that uses a dedicated memory region to allocate blocks of a fixed size. To enable device access to the region of shared memory where packet payload is allocated, we extend the libLXD and the LXD microkernel with support for the IOMMU interface. We configure the IOMMU to enable access to the packet payload region that is shared between the kernel and the LXD.

The ixgbe device driver uses system timers for several control plane operations. To provide timers inside LXDs, we rely on the timer infrastructure of the non-isolated kernel. Much like any other function pointer, we register the timer callback function pointer with the kernel. The callback caller stub sends an IPC to the isolated driver to trigger the actual callback inside the LXD. Finally, in the native driver, the NAPI polling function is invoked in the context of the softirq thread. We implement softirq threads as asynchronous threads dispatched from the LXD's dispatch loop.

The isolated dummy driver requires two cross-domain calls on the packet transmission path. The first call is invoked by the non-isolated kernel to submit the packet to the driver (ndo_start_xmit()), and the second is called by the driver after the packet was processed by the device and is ready to be released (consume_skb()). To reduce overheads of isolation, we introduce the "half-crossing" optimization. Specifically, for the functions that do not return a value, e.g., consume_skb() that releases the network packet to the kernel, we send the "call" message across the isolation boundary, but do not wait for the arrival of the reply message.

4.2 Multi-Queue Block Device Drivers

We implement a decomposed version of the nullblk block driver [4]. The nullblk driver is not connected to a real NVMe device, but instead emulates the behavior of the fastest possible block device in software.

Linux multi-queue block layer Linux implements a multi-queue (MQ) block layer [7] to support low-latency, high-throughput PCIe-attached non-volatile memory (NVMe) block devices. On par with network adapters, today NVMe is one of the fastest I/O subsystems in the kernel. To fully benefit from the asynchronous multi-queue layer, user-level processes rely on the new asynchronous block I/O interface that allows applications to submit batches of I/O requests to the kernel and poll for completion later. In the case of direct device access, the kernel performs all request processing starting from the system call to leaving the request ready for the DMA in the context of the same process that issued the io_submit() system call. The kernel returns to the process right after leaving request in the DMA ring buffers. Later the process polls for completion of the request by either entering the kernel again, or by monitoring a user-mapped page where the kernel advertises completed requests. Being allocated inside a user-level page, the pointer to the request is passed to the kernel, the kernel "pins" the page ensuring that it does not get swapped out while the request is in-flight. For each request the device driver adds the page containing the request to the IOMMU of the device, hence permitting the direct access to the request payload.

Decomposed block driver Similar to network drivers, we develop IDL specifications of the block driver interface, which consists of 68 lines of IDL code. The isolated nullblk driver requires three cross-domain calls on the I/O path. The first call is invoked by the non-isolated kernel to pass a block request from the block layer to the driver. The driver itself invokes two functions of the non-isolated kernel: blk_mq_start_request() and blk_mq_end_request(). The blk_mq_start_request() function passes the pointer to the request back to the block layer to inform it that request processing has started, and the I/O is ready to be issued to the device. The block layer now associates a timer with this particular request to ensure that if the completion for that request does not arrive in time, it can either abort the I/O operation or try to enqueue the request again. The blk_mq_end_request() allows the driver to inform the block layer that the request is completed by the device, and is ready to go up the block layer back to the user process.

We utilize ASYNC() and DO_FINISH() to implement an asynchronous loop on the submission path. A batch of requests is submitted by the application, hence we dispatch them to the nullblk LXD asynchronously. We provide a detailed analysis of isolation overheads in Section 5.6.

5 Evaluation

We conduct all experiments in the openly-available CloudLab network testbed [65].[3] We utilize CloudLab d820 servers with four 2.2 GHz 8-Core E5-4620 processors and 128 GB RAM. All machines run 64-bit Ubuntu 18.04 Linux with the kernel version 4.8.4. In all experiments we disable hyper-threading,

[3]LXDs are available at https://github.com/mars-research/lxds.

Operation	Cycles (Cycles per request)
Context switch	29-41
1 non-blocking ASYNC()	46
1 blocking ASYNC()	124
4 blocking ASYNC()s	374 (93.5)

Table 1: Overhead of asynchronous threads.

Operation	Cycles
seL4 same-core d820 (without PCIDs)	1005
seL4 same-core c220g2 (with PCIDs)	834
LXDs cross-core r320 (non-NUMA)	448
LXDs cross-core d820 (NUMA)	533

Table 2: Intra-core vs cross-core IPC.

turbo boost, and frequency scaling to reduce the variance in benchmarking.

5.1 Asynchronous Runtime

LXDs rely on the asynchronous runtime to hide the overheads of cross-domain invocations. To evaluate the effectiveness of this design choice, we conduct two sets of experiments that measure and compare overheads of asynchronous threads, and synchronous invocations.

Overhead of asynchronous threads We conduct four experiments that measure overheads of the asynchronous runtime (Table 1). In all tests we run 10M iterations and report an average across five runs. The first test measures the overhead of creating and tearing down a minimal asynchronous block of code that just increments an integer, but does not block. Each iteration takes 46 cycles which includes allocating and deallocating a stack for the new thread, and two stack switches to start and end execution of the thread. In the second test we measure the overhead of switching between a pair of asynchronous threads that takes 29 cycles and uses a sequence of 20 CPU instructions. Out of 20 instructions 16 are memory accesses that touch the first level cache and take two cycles each [28] (six instructions are required to save and restore callee saved registers, and two save and restore instruction pointer and stack registers). If, however, the context switch touches additional metadata, e.g., adds the thread to the runqueue, the overhead of the context switch grows to 41 cycles due to additional memory accesses.

The third and fourth tests measure the overhead of executing one and four blocking ASYNC() code blocks, i.e., each thread executes yield() similar to the IPC path. The overhead of creating one blocking asynchronous block (third test in Table 1) is 124 cycles, which consist of the cost to create and tear down a non-blocking asynchronous thread (46 cycles) and three context switches required to block and unblock the thread, and switch back to the main thread when the DO_FINISH() block is reached. If, however, we execute four asynchronous blocks in a loop the total overhead comes to 374 cycles or 93.5 cycles per one asynchronous block. Overall, we conclude that asynchronous threads are fast, and come close to the speed of manual management of pending invocations in a message-passing system.

Batch size	Cycles (cycles per msg)	
	Manual	ASYNC()
1	533	568
4	876 (219)	1111 (277)
8	1262 (157)	2096 (262)

Table 3: Benefits of manual and ASYNC() batching.

5.2 Same-core vs cross-core IPC

Same-core IPC To understand the benefits of cache-coherent cross-core invocations over traditional same-core address-space switches, we compare LXDs' cross-core channels with the synchronous IPC mechanism implemented by the seL4 microkernel [26]. We choose seL4 as it implements the fastest synchronous IPC across several modern microkernels [55]. As d820 servers do not provide support for tagged TLBs (PCIDs) that improve IPC performance by avoiding an expensive TLB flush on the IPC path, in addition to the d820 machines we report results for the same IPC tests on an Intel E5-2660 v3 10-core Haswell 2.6GHz machine (CloudLab c220g2 server) that implements support for tagged TLBs. To defend against Meltdown attacks, seL4 provides support for a page-table-based kernel isolation mechanism similar to KPTI [37]. However, this mechanism negatively affects IPC performance due to an additional reload of the page table root pointer. Since recent Intel CPUs address Meltdown attacks in hardware, we configure seL4 without these mitigations. On d820 machines without PCIDs support, seL4 achieves the median IPC latency of 1005 cycles (Table 2). On the c220g2 servers with tagged TLBs enabled the IPC latency drops to 834 cycles (Table 2).

Cross-core IPC To measure the overhead of cross-core cache-coherent invocations, we conduct a minimal call/reply test in which a client thread repeatedly invokes a function of a server via an LXD's asynchronous communication channel. Client and server are running on two cores of the same CPU socket. Since multi-socket NUMA machines incur higher cache-coherence overheads and thus have slower cross-domain invocations, in our experiments we a NUMA and a non-NUMA machine with a similar CPU: a four socket d820 NUMA server and a single-socket non-NUMA r320 CloudLab server configured with one 2.1 GHz 8-core Xeon E5-2450 CPU. In all experiments we run 100M call/reply invocations and report an average across five runs (Table 2). On a non-NUMA r320 machine, cross-core IPC takes 448 cycles. On a NUMA d820 machine, this number increases to 533 cycles.

Two additional observations are important. First, communication between hardware threads of the same CPU core takes less time than communication between cores (we measure the overhead of cross-core invocations to be only 105 cycles on the non-NUMA r320 machine and 133 cycles on the NUMA

d820). Typically, however, a single LXD serves requests from multiple cores of the monolithic kernel, and hence only a single core can benefit from proximity to the logical core.

Second, communication outside of the NUMA node incurs high overheads due to crossing inter-socket links. On the d820 server, a cross-socket call/reply invocation takes 1988 cycles over one inter-socket hop, which is higher than overhead of a synchronous same-core IPC. Note that on a batch of 4 and 8 this number drops to 900 and 535 cycles per message respectively. We anticipate that each NUMA node will run a local LXD thread and hence the crossings of NUMA nodes will be rare (this design makes sense as high-throughput isolated subsystems, e.g., network and NVMe drivers, are CPU-bound and anyway require multiple LXD threads to keep up with invocations from multiple kernel threads).

Finally, we make an observation that compared to overheads of synchronous IPC invocations (both on the same core and cross-core) the overheads of asynchronous threads is relatively small (93.5 cycles per-request in a batch of four (Table 1)). Therefore, the use of asynchronous threads for batching and pipelining of multiple cross-domain invocations is justified.

5.3 Message Batching

To evaluate the benefits of aggregating multiple cross-core invocations in a batch, we conduct an experiment that performs call/reply invocations in batches of messages ranging from 1 to 8. On a batch of 4 messages a call/reply invocation takes only 876 cycles, or 219 cycles per invocation on a d820 NUMA machine (Table 3). On a batch of 8 messages the overhead per one call/reply invocation drops to 157 cycles per message. For a batch of messages, the cross-core IPC sends call/reply invocations through independent cache lines. The CPU starts sending the next message right after issuing loads and stores to the hardware load/store queue, but without waiting for completion of the cache-coherence requests effectively pipelining multiple outstanding cache coherence requests.

Composable batching with ASYNC() Finally, we analyze how cross-core invocations are affected if the batches of messages are created by blocking asynchronous threads instead of the manual, message-passing style batching we analyzed above. To evaluate overheads of asynchronous threads, we design an IPC test that performs a series of cross-core function invocations from inside an ASYNC() block (Table 3). We run a loop of length 1, 4, and 8. The body of the loop is an asynchronous code block that invokes a function on another core. Instead of waiting for the reply, each asynchronous thread yields and continues to the next iteration of the loop that dispatches the new asynchronous thread. For the loop of length 1, 4, and 8, compared to the manual batch, ASYNC() introduces overhead ranging from 35 cycles on a batch of one to 105 cycles per message on a batch of 8 (Table 3).

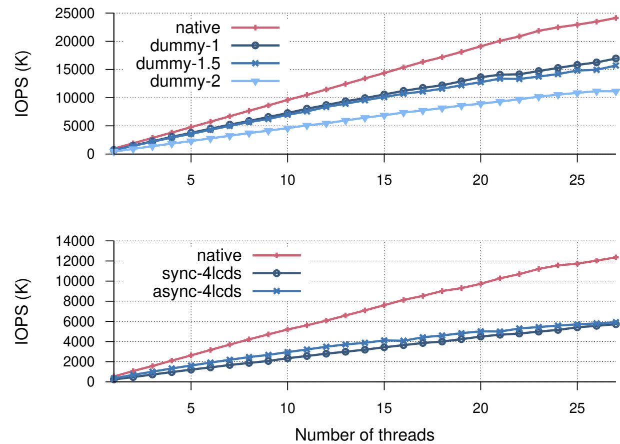


Figure 3: Performance of the dummy driver

5.4 Dummy Device Driver

We utilize the dummy driver as a platform for several benchmarks that highlight overheads of isolation in the context of a "fast" device driver (dummy is a good, representative example of such device driver as it serves an infinitely fast device and is accessed through a well-optimized I/O submission path of the kernel network stack). In all experiments we use the iperf2 benchmark that measures the transmit and receive bandwidth for different payload sizes, and run the tests on d820 servers (Figure 3). We configure the isolated dummy driver with a varying number of cores ranging from one to four in such a manner that one LXD thread runs on each socket of the 4-socket d820 system. Specifically, on a 32-core system, the isolated dummy can support up to 27 iperf threads (i.e., four cores of the system are dedicated to LXD threads, and one core is occupied by the kLXD thread servicing control plane invocations from the LXD). We assign the first six iperf threads to the first socket (one core of the CPU socket is occupied by the LXD thread and one by the kLXD thread), then we assign the next seven iperf threads (7-13) to the next socket, and so on (Figure 3). We report the total number of device driver I/O requests per-second (IOPS) across all threads (we report an average across five runs on the maximum transmission unit (MTU) size packets).

In our first experiment we change dummy to perform only *one* crossing between the kernel and the driver for sending each packet (`dummy-1`, Figure 3). This synthetic configuration allows us to analyze overheads of isolation in the ideal scenario of a device driver that requires only one crossing on the device I/O path. With one application thread the non-isolated driver achieves 956K IOPS (i.e., on average, a well-optimized network send path takes only 2299 cycles to submit an MTU-sized packet from the user process to the network interface). The isolated driver achieves 730K IOPS (76% of the non-isolated performance), and on average requires 3009 cycles to submit one packet. Of course, the isolated driver utilizes one extra core for running the LXD. Isolation adds an overhead of

710 cycles per-packet, which includes the overhead of the IPC and processing of the packet by the driver (in this experiment, LXDs do not benefit from any asynchrony; all packets are submitted synchronously). On 27 threads the isolated driver achieves 70% of the performance of the non-isolated driver. Compared to the configuration with one application thread, the slight drop in performance is due to the fact that each of four LXDs service up to seven application threads which adds overhead to the LXD's dispatch processing loop.

In practice, the dummy driver requires two domain crossings for submitting each packet (Section 4). We evaluate how performance of isolated drivers degrades with the number of crossings by running a version of dummy that performs two full cross-domain invocations (`dummy-2`, Figure 3). On one thread, two crossings add overhead of 1794 cycles per packet. The "half-crossing" optimization, however, reduces the overhead of two crossings from 1794 cycles per-packet to only 814 cycles (`dummy-1.5`, Figure 3).

Asynchronous threads To evaluate the impact of asynchronous communication, we perform the same iperf2 test with a packet size of 4096 bytes. When the packet size exceeds MTU, the kernel fragments each packet into MTU-size chunks suitable for transmission and submits each chunk to the driver individually. In general, multiple domain crossings caused by fragmentation negatively affect performance of the isolated driver. We compare three configurations: a non-isolated dummy driver (`native`, Figure 3), a synchronous version of LXDs (`sync-4-lcds`) and asynchronous version that leverages ASYNC() to invoke the driver in a parallel loop (`async-4-lcds`). Configured with one iperf thread, a non-isolated driver achieves 534K IOPS, i.e., on average it requires 4114 cycles to submit a 4096 byte packet split in three fragments. Performance of the synchronous version of the isolated driver is heavily penalized by the inability to overlap communication, i.e., waiting for LXD replies, and processing of further requests. The synchronous version achieves only 236K IOPS (44% of non-isolated performance). The asynchronous isolated driver is able to benefit from pipelining of three fragmented packets with asynchronous threads (it achieves 341K IOPS or 63.8% of non-isolated performance). Note, that as the number of application threads grows, the benefits of asynchronous threads gradually disappear. With 27 iperf threads both synchronous and asynchronous configurations achieve similar performance (36% and 37% of the native driver respectively). As the number of application threads increases, each core of the isolated driver that processes requests from up to seven iperf threads becomes heavily utilized. Each LXD thread dispatches kernel invocations in a round-robin manner from a set of cross-core communication channels. If all channels are active, the performance of each iperf thread is dominated by the time spent waiting for its turn to be processed by the LXD. On a batch of only three messages, asynchronous threads do not provide sufficient benefits to tolerate this latency.

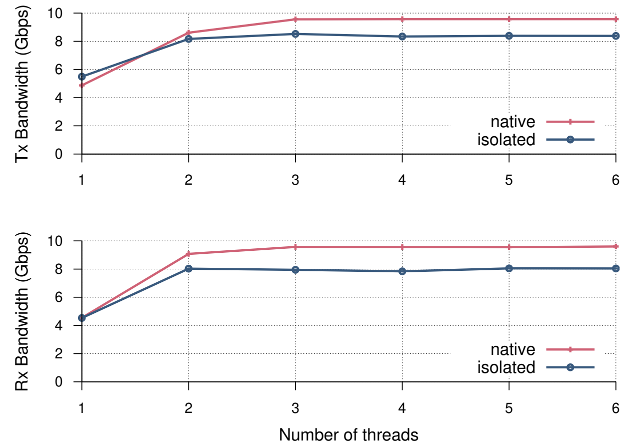


Figure 4: Ixgbe Tx and Rx bandwidth.

5.5 Ixgbe Device Driver

To measure performance of the isolated ixgbe driver, we configure an iperf2 test with a varying number of iperf threads ranging from one to six (Figure 4). On our system, a small number of application threads saturates a 10Gbps network adapter. Configured with one iperf thread, on the MTU size packet the isolated ixgbe is 12% faster compared to the isolated system on the network transmit path, although at the cost of using an extra core. This advantage disappears as the LXD becomes busy handling more than one iperf thread. Nevertheless, from three to seven threads, the isolated driver stays within 6-13% of the performance of the native device driver which saturates the network interface with three and more application threads.

On the receive path, the isolated driver is 1% slower for one application thread. Two factors attribute to performance of the isolated driver: 1) it benefits from an additional core, and 2) it uses asynchronous threads for NAPI polling instead of native threads used by the Linux kernel for handling IRQs. Asynchronous threads provide a faster context switch compared to the native Linux kernel threads. Similar to transmit path, this advantage disappears with larger number of application threads. From two to six threads the isolated driver stays within 12-18% of the performance of the native driver.

To measure the end-to-end latency, we rely on the UDP request-response test implemented by the netperf benchmarking tool. The `UDP_RR` measures the number of round-trip request-response transactions per second, i.e., the client sends a 64 byte UDP packet and waits for the response from the server. The native driver achieves 26688 transactions per second (which equals the round-trip latency of 40μs), the isolated driver is 7% (2.6μs) faster with 24975 transactions per second (round-trip latency of 37.4μs). Again the isolated driver benefits from a faster receive path due to low-overhead context switch of asynchronous threads. As the network is lightly loaded during the latency test even with six application threads the isolated driver remains 3.4μs faster achieving the latency

of 43.4μs versus 46.8μs achieved by the native driver.

5.6 Multi-Queue Block Device Driver

In our block device experiments, we use fio to generate I/O requests. To set an optimal baseline for our evaluation, we chose the configuration parameters that provide the lowest latency path to the driver, so that overheads of isolation are emphasized the most. We use fio's libaio engine to overlap I/O submissions, and bypass the page cache by setting direct I/O flag to ensure raw device performance. Similar to dummy, in isolated configuration, the nullblk LXD fully utilizes one extra core on every CPU socket. We run the same configurations as for dummy, e.g., one LXD thread on each NUMA node. We placing the first six fio threads on the first NUMA node, next seven fio threads on the second NUMA node, and so on, up until 27 fio threads. We vary the number of fio threads from 1 to 27 and report results for two block sizes—512 bytes and 1MB—which represent two extreme points: a very small and a very large data block. For each block size, we submit a set of requests at once ranging the number of requests from 1 to 16 and then poll for the same number of completions. Since the nullblk driver does not interact with an actual storage medium writes perform as fast as reads, hence we utilize read I/O operations in all experiments.

The native driver achieves 295K IOPS for the packet size of 512 bytes and the queue of one (Figure 5). In other words, a single request takes about 7457 cycles to complete. The isolated driver achieves 235K IOPS (or 79% of non-isolated performance). The isolation incurs an overhead of 1904 cycles due to three domain crossings on the critical path. For a queue of 16 requests, the isolated driver benefits from asynchronous threads which allow it to stay within 4% of the performance of the native driver for as long as it stays in one NUMA node (from 1 to 6 fio threads). Both native and isolated drivers suffer from NUMA effects due to the fact that Linux block layer collects performance statistics for every device partition. The blk_mq_end_request() function acquires a per-partition lock and updates several global counters. The native driver faces performance drops when it spills outside of a NUMA node at 9, 17, and 25 fio threads (Figure 5). The isolated driver experiences similar drops at 7, 14, and 21 fio threads. On the block size of 1M, inside one NUMA node the isolated driver stays within 10% of the performance of the native driver for both queues of one and 16 requests. Outside of one NUMA node the performance of both native and isolated drivers suffers from NUMA effects. We speculate that NUMA degradation can be fixed by changing the kernel to use per-core performance counters [10].

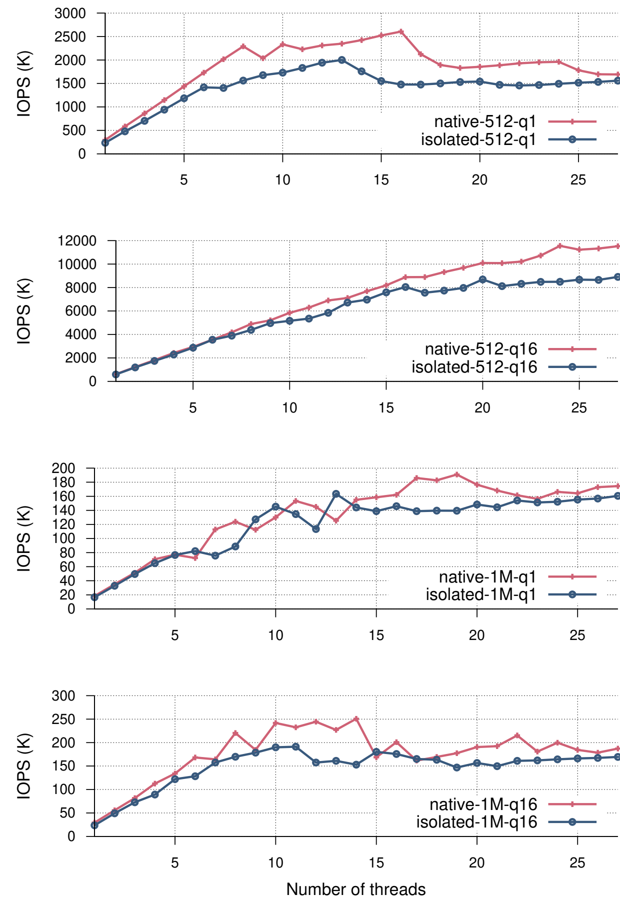


Figure 5: Performance of the nullblk driver

6 Conclusions

LXDs provide general abstractions and mechanisms for isolating device drivers in a full-featured operating system kernel. By employing several design choices—relying on an asynchronous execution runtime for hiding latency of cross-domain invocations, developing general decomposition patterns, and relying on cross-core invocations—we demonstrate the ability to isolate kernel subsystems with tightest performance budgets. We hope that our work will gradually enable kernels to employ practical isolation of most device drivers and other kernel subsystems that today account for the majority of the kernel code.

Acknowledgments

We thank ASPLOS 2018, OSDI 2018, and USENIX ATC 2019 reviewers and our shepherd, Andrew Baumann, for in-depth feedback on earlier versions of the paper and numerous insights. Also we would like to thank the Utah Emulab and CloudLab team, and especially Mike Hibler, for his continuous support and endless patience in accommodating our hardware requests. This research is supported in part by the National Science Foundation under Grant Numbers 1319076, 1527526, and 1817120 and Google.

References

[1] Code-Pointer Integrity in Clang/LLVM. https://github.com/cpi-llvm/compiler-rt.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J.

Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference (ATC)*, pages 289–302, Berkeley, CA, USA, 2002.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139(140):116, 2005.

[4] Jens Axboe. Null block device driver. https://www.kernel.org/doc/Documentation/block/null_blk.txt, 2019.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44, New York, NY, USA, 2009.

[6] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, 1991.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 22:1–22:10, New York, NY, USA, 2013.

[8] 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 (ASIACCS)*, pages 30–40, 2011.

[9] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Berkeley, CA, USA, 2010.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX Annual Technical Conference (ATC)*, pages 9–22, 2010.

[12] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[14] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[15] Citrix. XenClient. http://www.citrix.com/products/xenclient/how-it-works.html.

[16] 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 *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 189–202. ACM, 2011.

[17] Microsoft Corporation and Digital Equipment Corporation. The component object model specification, 1995.

[18] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, and Jonathan Walpole. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[19] CVE Details. Vulnerabilities in the Linux kernel by year. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[20] CVE Details. Vulnerabilities in the Linux kernel in 2018. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/Linux-Linux-Kernel.html.

[21] Data61 Trustworthy Systems. *seL4 Reference Manual*, 2017. http://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

[22] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 33–48. ACM, 2013.

[23] Distributed component object model (DCOM) remote protocol specification. https://msdn.microsoft.com/library/cc201989.aspx.

[24] DDEKit and DDE for Linux. http://os.inf.tu-dresden.de/ddekit/.

[25] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *ACM SIGPLAN Notices*, volume 32, pages 44–56. ACM, 1997.

[26] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 133–150. ACM, 2013.

[27] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture*. Technische Universität, Dresden, Fakultät Informatik, 2007.

[28] Agner Fog. Instruction tables. http://www.agner.org/optimize/instruction_tables.pdf.

[29] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 38–51, 1997.

[30] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[31] Linux FUSE (filesystem in userspace). https://github.com/libfuse/libfuse.

[32] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[34] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114. ACM, 2000.

[35] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 2002.

[36] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2008.

[37] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[38] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, October 22 2000.

[39] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. AC: composable asynchronous IO for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

[40] Härtig, H. Security architectures revisited. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 16–23. ACM, 2002.

[41] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[42] Herder, J.N. and Bos, H. and Gras, B. and Homburg, P. and Tanenbaum, A.S. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[43] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[44] Nick Holloway. Dummy net driver. https://elixir.bootlin.com/linux/latest/source/drivers/net/dummy.c, 1994.

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

[46] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1868–1882, New York, NY, USA, 2018. ACM.

[47] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–48, 2016.

[48] Antti Kantee. *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. PhD thesis, 2012.

[49] Vinay Katoch. Whitepaper on bypassing ASLR/DEP. http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf, 2011.

[50] Kenton Varda. Cap'n Proto Cerealization Protocol. http://kentonv.github.io/capnproto/.

[51] Kil3r and Bulba. Bypassing StackGuard and Stack-Shield. *Phrack Magazine*, 53, 2000.

[52] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *USENIX Annual Technical Conference (ATC)*, pages 7:1–7:14, 2007.

[53] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[54] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *Workshop on I/O Virtualization*, 2011.

[55] 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*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.

[56] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[57] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270. IEEE, 2009.

[58] Tilo Müller. ASLR smack and laugh reference. *Seminar on Advanced Exploitation Techniques*, 2008.

[59] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, 2001.

[60] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 116–132, New York, NY, USA, 2013.

[61] Object Management Group. OMG IDL Syntax and Semantics. http://www.omg.org/orbrev/drafts/3_idlsyn.pdf.

[62] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[63] Jon Rafkind. Vembyr - multi-language PEG parser generator written in Python, November 2011. http://code.google.com/p/vembyr/.

[64] Matthew J Renzelmann and Michael M Swift. Decaf: Moving device drivers to a modern language. In *USENIX Annual Technical Conference (ATC)*, 2009.

[65] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:*, 39(6):36–38, December 2014.

[66] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[67] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–8, 2010.

[68] Green Hills Software. INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[69] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107. ACM, 2002.

[70] Hajime Tazaki. An introduction of library operating system for Linux (LibOS). https://lwn.net/Articles/637658/.

[71] Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59, 2002.

[72] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.8, update 3. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[73] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog*, 2008.

[74] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[75] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondrian memory protection. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 31–44. ACM, 2005.

[76] Jonathan Woodruff, Robert NM 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 *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[77] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Usenix Security Symposium*, 2018.

JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre

Nadav Amit
VMware Research

Fred Jacobs
VMware

Michael Wei
VMware Research

Abstract

The Spectre family of security vulnerabilities show that speculative execution attacks on modern processors are practical. Spectre variant 2 attacks exploit the speculation of indirect branches, which enable processors to execute code from arbitrary locations in memory. Retpolines serve as the state-of-the-art defense, effectively disabling speculative execution for indirect branches. While retpolines succeed in protecting against Spectre, they come with a significant penalty — 20% on some workloads.

In this paper, we describe and implement an alternative mechanism: the JumpSwitch, which enables speculative execution of indirect branches on safe targets by leveraging indirect call promotion, transforming indirect calls into conditional direct calls. Unlike traditional inlining techniques which apply call promotion at compile time, JumpSwitches aggressively learn targets at runtime and leverage an optimized patching infrastructure to perform *just-in-time* promotion without the overhead of binary translation.

We designed and optimized JumpSwitches for common patterns. If a JumpSwitch cannot learn safe targets, we fall back to the safety of retpolines. JumpSwitches seamlessly integrate into Linux, and are evaluated in Linux v4.18. We show that JumpSwitches can improve performance over retpolines by up to 20% for a range of workloads. In some cases, JumpSwitches even show improvement over a system without retpolines by directing speculative execution into conditional direct calls just-in-time and reducing mispredictions.

1 Introduction

Spectre is a family of security vulnerabilities first disclosed to the public in January 2018 [38]. Spectre manipulates a processor to speculatively execute code which leaves side-effects that can be observed even after a processor decides the speculation is incorrect and reverses its execution. Spectre attacks have shown that it is practical to extract data which leaks from these observable side-effects, known as side-channels. As a result, both processor manufacturers and software developers have sought to defend against Spectre-style attacks by restricting speculative execution.

In order to execute a Spectre attack, malicious code must be able to control what the processor speculatively executes. While several vectors of attack exist, indirect branches, which enable a processor to execute code from arbitrary locations in memory have been shown to be especially vulnerable. Variant 2 of Spectre, known as "Branch Target Injection" [38], leverages indirect branches by causing the processor to mispredict the branch target and speculatively execute code designed to leave side-effects. To mitigate against this attack, a code sequence known as a *retpoline* [49] was developed, which is used in software and throughout the OS kernels today. Retpolines work by effectively disabling speculative execution of indirect branches. Instead of allowing the processor to speculate on the target of an indirect branch, retpolines force the processor to speculatively execute an infinite loop until the target is known. This is achieved by leveraging return (or `RET`) instructions, which are speculatively executed using a different mechanism than indirect branches.

While retpolines are effective in mitigating against Spectre variant 2, preventing speculative execution of indirect branches comes at a significant cost: in some benchmarks, we observe up to a 20% slowdown due to retpolines, making Spectre variant 2 one of the most expensive Spectre attacks to defend against. Unfortunately, indirect branches are used extensively in software: they are the basis for software indirection, they enable object-oriented constructs such as virtual functions and they are present throughout the OS kernels to enable their modular design. Spectre has made optimizing indirect branches in software more important than ever, since Spectre vulnerable hardware is no longer able to do so.

This paper describes our implementation of an alternative mechanism, the JumpSwitch, which leverages a technique known as indirect call promotion to transform indirect calls into conditional direct calls. Indirect call promotion is often employed by compilers [6, 9] to take advantage of profiling data, or binary translators and JIT engines to avoid expensive lookups in code caches [20]. These techniques alone, however, are insufficient for OS kernels. Mechanisms such as kernel address space layout randomization (KASLR) [45] dynamic modules and JITed code (e.g., eBPF) make it difficult to reliably determine branch targets. Compiler-based approaches such as PDO/FDO "lock" the results in a binary, requiring recompilation for best performance and preventing regressions [40] when the workload changes [54], which may be untenable as most distributions package kernels as binaries. Binary translation of the kernel incurs significant overheads

while restricting functionality, obviating the benefits of indirect call promotion [27].

Building on this observation, JumpSwitches learn execution targets either statically at compile-time or dynamically at runtime and promote indirect calls using *just-in time promotion*, learning targets using a lightweight mechanism as the kernel runs. If correct targets cannot be learned, JumpSwitches fall back to using retpolines. JumpSwitches are fully integrated with the Linux kernel's build system.[1] Through the Linux build infrastructure, we are able to seamlessly update the kernel to leverage JumpSwitches without source code changes. However, to further improve performance, we provide more advanced JumpSwitches known as *semantic JumpSwitches* which are able to take advantage of the rich semantic information available within the kernel such as process tables, and we provide five different types of JumpSwitches tailored for different use cases.

Our evaluation of JumpSwitches shows a performance improvement over retpolines in a variety of workloads by up to 20%. In some cases, we are able to even show improvement on a system without retpolines, so JumpSwitches are useful even if Spectre variant 2 is mitigated in hardware. Since JumpSwitches are implemented within the kernel, they may potentially benefit every workload running on Linux.

While conditional direct calls are vulnerable to Spectre variant 1 ("Bounds Check Bypass") [30,39,49], another Spectre vulnerability which retpolines do defend against, other less expensive mechanisms which JumpSwitches are compatible with, such as static analysis and selective masking/serialization are typically used to defend against Spectre variant 1 [12]. Notably, future "Spectre safe" hardware which do not need retpolines will still be vulnerable to Spectre variant 1 [46] and require Spectre variant 1 mitigations. JumpSwitches provide the same level of safety as this future hardware.

This paper makes the following contributions:

- We explore Spectre and the impact of current and future mitigations on the performance of indirect calls. (§2.1)
- We explore previous work for indirect call promotion and describe how JumpSwitches revisit indirect call promotion in the era of Spectre. (§2.2, §2.4).
- We describe our implementation of JumpSwitches (§3) which features:
  - Five types of JumpSwitches, which are optimized for the common case (generic JumpSwitches) and special use cases (semantic JumpSwitches) (§3.1).
  - A mechanism for learning which updates JumpSwitches outside the path of execution (§3.2).
  - A integration with the Linux kernel which leverages semantic information within the kernel (system calls, process information, `seccomp`) (§3.4).
- We evaluate JumpSwitches and show that they improve performance in real-world benchmarks by up to 20% (§4).
- We conclude with remarks about the future of speculative execution (§5).

[1] We have submitted upstream patches to the Linux kernel, with positive feedback from the Linux community.

2 Indirect Branches

Indirect branches are a basic type of processor instruction which enable programs to dynamically change what code is executed by the processor at runtime. To execute an indirect branch, the processor computes a *branch target*, which determines what instruction to execute after the indirect branch. Since targets are dynamically computed, indirect branches can execute code depending on what data is present. This indirection is most frequently used to enable polymorphism. For example, in C++, different functions may be executed depending on an object's type, or in Linux, different functions may be executed depending on whether an address is IPv4 or IPv6. To enable this functionality, a compiler will generate an indirect branch which computes the correct function as a branch target based on the object's type. Indirect branches are not limited to object-oriented code: they are used by many constructs such as callbacks and jump tables as well.

The use of indirect branches, however, poses a significant problem for modern processors which are heavily pipelined and require a constant instruction stream to achieve maximum performance. With an indirect branch, the processor cannot fetch the next instruction until the branch target is computed, resulting in pipeline stalls that significantly degrade performance. To eliminate these stalls, processors leverage *speculation*, which guesses the branch target and rolls back executed instructions if the guess is later determined to be incorrect. In many processors, speculation is done through a *branch target buffer* (BTB), which serves as a cache for previous branch targets of indirect branches.

The Spectre family of attacks observed that rolling back speculative execution can be incomplete. For example, speculative execution of privileged code could leave memory as a result of that execution in the processor's caches. Timing attacks can be used by an unprivileged process to determine what memory was fetched, resulting in a data leak. Spectre variant 2, known as "Branch Target Injection" specifically targeted indirect branches and the BTB [30,38]. In variant 2, either training the predictor to speculatively execute a malicious branch target or causing a collision in the BTB (which uses a subset of address bits for performance) resulting in an indirect branch speculatively executing code which leaves side-effects in the processor cache. This work focuses on variant 2, and when we refer to Spectre we refer to variant 2 of the attack.

In the next sections, we discuss both current and future mitigations for Spectre and how performance is affected. We then discuss software mechanisms currently used to optimize indirect branches, and present how JumpSwitches enable dynamically optimization of indirect branches while mitigating

```
1  call %rax          call  rax_retpoline

2                  rax_retpoline:
3                      call  target
4                  capture: pause
5                      lfence
6                      jmp  capture
7                  target: mov %rax, [%rsp]
8                      ret
```

Figure 1: Retpolines. Unsafe indirect branches on the left are replaced with a call (line 1) to a retpoline thunk (lines 2-8).

Spectre style attacks. For the rest of this paper, we focus on the Intel x86 architecture. We believe that JumpSwitch can be applied to other Spectre vulnerable architectures, such as AMD, ARM and IBM.

2.1 Spectre Mitigations

Mitigations for Spectre can be categorized into three general categories: software, hardware microcode and hardware.

Software mitigations. Retpolines [49] are the currently preferred method for mitigating Spectre. Retpolines, shown in Figure 1, work by directing speculative execution of indirect branches into an infinite loop. This is achieved by redirecting indirect branches to a thunk which captures the branch target on the return stack buffer (RSB) and uses a `RET` instruction instead of a `CALL`. A `RET` uses a different speculative prediction mechanism which leverages the RSB rather than the vulnerable BTB, so it does not suffer from the same vulnerability as a `CALL` instruction. Speculative execution of the return path will execute lines 4-6, labeled `capture`, which exhausts speculative execution by looping forever. The `PAUSE` instruction on line 4 is used to release processor resources to a hardware simultaneous multithread (SMT) or to save power if no execution is needed, and the `LFENCE` instruction acts as a speculation barrier [15, 36].

Retpoline safety is based on the behavior of the predictor for the `RET` instruction. On Intel architectures before Skylake, speculation for `RET` instructions were always computed from the RSB [14]. On Skylake and after however, the contents of the BTB may be used if the RSB is empty, making these architectures vulnerable even if retpolines are used [31]. To prevent an attack which causes the underflow of the RSB, a technique known as RSB stuffing can be used on these architectures on events which could cause the RSB to be emptied [51]. However, a deep call stack may still result in unpredictable overflows, leaving these architectures vulnerable even with both RSB stuffing and retpolines enabled.

Measuring precise performance overheads of retpolines is difficult because they affect deep microarchitectural state such as the RSB, and may cause future mispredictions. Empirically, retpolines have been observed to result in as much as a 20% slowdown on some workloads, increasing the cost of an indirect branch from a worst-case cost of around 10 cycles [13] (indirect branch misprediction) to almost 70 cycles in the common case. The performance penalty Spectre imposes on indirect branches makes optimizing them much more important.

Hardware microcode mitigations. As a result of Spectre, microcode updates have been introduced which add functionality to control indirect branch predictions. These mechanisms include Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Predictors (STIBP) and Indirect Branch Predictor Barrier (IBPB) [14]. IBRS works by defining four privilege levels: host and guest modes with corresponding user and supervisor modes. The processor guarantees that lower privilege modes and other logical processors cannot control the predictors of more privileged modes. STIBP and IBPB are used to protect VMs and processes across context switches by preventing predictions in one thread or context from affecting another thread or context.

These microcode mechanisms can replace retpolines. Unfortunately, the performance penalty of IBRS is quite high: Since it is implemented in microcode, it likely flushes the BTB on each privilege transition and requires a model-specific register (MSR) write whenever a privilege transition occurs. IBRS has an extremely high overhead compared to retpolines (observed to be as high as 25–53%+) [17, 47, 50].

IBRS is the only mechanism which completely protects Skylake and future processors, because the BTB may be used if the RSB underflows. However, due to concerns with complexity and performance, the Linux kernel has adopted the use of retpolines on all vulnerable architectures instead, using RSB stuffing to protect the RSB [51, 53].

Hardware mitigations. Finally, Intel has proposed mitigations for Spectre in new microarchitectures which require new microprocessors to be deployed. These mitigations cannot address Spectre in the billions of already deployed processors [37]. This includes technologies such as Enhanced Indirect Branch Restricted Speculation (Enhanced IBRS) [16] and control flow enforcement technology (CET) [14]. Enhanced IBRS eliminates the overhead of IBRS by removing the requirement to write to an MSR. CET restricts the speculations which the processor can make by requiring that indirect branches target only special `ENDBR` (end-branch) instructions.

Hardware mitigations will be much more performant than software or hardware microcode mitigation today, since future processors will be able to make deep changes to the microarchitecture to account for Spectre. Practically implementing these technologies on code which must run on both secure and vulnerable hardware remains an open question, especially

in the cloud where live migration between processors remains a real possibility [36,48].

2.2 Indirect Call Promotion

The cost of indirect calls have been historically observed to be high, even before the discovery of Spectre. The most common technique used to reduce the cost of indirect calls is known as *indirect call promotion*, where likely call targets are promoted to conditional direct calls, reducing the chance that the processor will mis-speculate an indirect call using a code fragment similar to that shown in Figure 2 [6,7].

Promoting indirect calls comes at a cost, however. Each promoted call increases code size, which significantly decreases performance if infrequently used targets are promoted. Code size also limits the number of targets that can be promoted. Indirect branches which call a large number of targets with equal likelihood are poor candidates for promotion.

Compilers can determine and promote likely targets at compile-time [9,23], through the use of profile-guided optimization or feedback-directed optimization (PGO/FDO) [10, 11,26,32,41,42]. Collection of targets in the Linux kernel, which is the focus of this paper, is complicated by several factors. First, branches collected through the use of profiling may not resemble the actual execution at runtime and incorrect learned targets may even result in regressions [40], particularly if the code is executed under diverse conditions. Previous work has shown that simply executing the Linux kernel under different workloads results in different profiles [54,55]. Second, mechanisms such as KASLR [45], JITed code (e.g., eBPF programs) and dynamically loaded modules, may make it impossible to learn targets until runtime. Finally, incorporating learned branch targets requires recompilation, which may mean having different binaries for each workload or a deployment infrastructure which adapts binaries over time [10,42]. This is incompatible with how kernels are provided in most OS distributions: as binary packages.

While binary translators [18, 27, 28] and JIT compilers [25, 33, 44] are capable of collecting branch targets at runtime, binary translators result in a significant overhead as they are designed to translate and instrument the entire binary, and may need to restrict the kernel's functionality for best performance [27]. JIT compilers cannot directly run the Linux kernel, although some progress has been made to leverage LLVM to run applications as complex as Linux [3].

2.3 Alternative Solutions

After we released parts of our code [4], several solutions that employ indirect call promotion have been proposed by Linux developers. Hellwig added a fast-path to code that invokes DMA operation based on the I/O memory management unit (IOMMU) that the system uses. This solution effectively performs indirect branch promotion of each call to a single static predetermined target: the functions that are used when no IOMMU is used. As a result, the solution does not provide any benefit when an IOMMU is used [21].

A similar solution was introduced by Abeni to reduce the overhead of indirect branches in the network stack [2]. This solution is also static, requiring the developer to determine at development time what the likely targets are of each call. In addition, this solution is not suitable for calls to functions in loadable modules, whose address is unknown at compilation time. The implementation, which requires changing function calls into C macros reduces code readability. As a result this solution is limited to certain use-cases.

Finally, Poimboeuf [43] proposed a mechanism that addresses indirect calls whose target rarely changes. The proposed solution uses direct calls, and when the target changes, performs binary rewriting to change the direct call target. As this mechanism does not have a fallback path, it is only useful in certain calls. In addition, it requires the programmer to explicitly invoke binary rewriting and the use of C macros negatively affects code readability.

```
1  call %rax  |        cmp ${likely},%eax
2             |        jnz miss
3             |        call {likely}
4             |        jmp done
5             |  miss: call %eax
6             |  done:
```

Figure 2: Indirect call promotion. Indirect branches on the left are replaced with a direct branch (line 3) which is called if the prediction is correct at runtime (lines 1, 2). Otherwise, a normal indirect call is made (line 5).

2.4 JumpSwitches

JumpSwitches are designed to mitigate Spectre by taking advantage of indirect call promotion and leveraging semantic information not available by compile-time. JumpSwitches are motivated by the following observations:

- Retpolines induce ≈70 cycles of overhead which can be leveraged by software to resolve indirect branches.
- Promoting indirect branches can be combined with ensuring the safety of speculative branch execution.
- Committing to specific optimizations at compile-time can limit performance and potentially risk safety (such as assuming Spectre-safe hardware).

Threat Model. JumpSwitches assume that only indirect branches and returns are vulnerable to Spectre variant 2, as stated by Intel [14] and the original Spectre disclosure [30,38]. It is also to important to note that while retpolines can serve as a defense against Spectre variant 1 ("Bounds Check Bypass") [30,39,49], JumpSwitches do not defend against Spec-

Mechanism	Overhead	Spectre-Safe	Learning
retpolines [49]	medium	yes	none
EIBRS [48]	low	yes	none
PGO/FDO [54]	low	w/retpoline	static
DBT/JIT [27]	high	w/retpoline	dynamic
JumpSwitches	low	dynamic	dynamic+semantic

Table 1: Comparison of various indirect call mechanisms, their relative overheads and if they are Spectre-safe.

tre variant 1. We assert that replacing retpolines with JumpSwitches does not make the Linux kernel vulnerable to Spectre variant 1, as retpolines are used solely as a Spectre variant 2 defense. In Linux, static analysis and selective masking/serialization is used to defend against Spectre v1 [12], since Spectre variant 1 defenses are still necessary even in "Spectre safe hardware", where retpolines are turned off for performance. Spectre variant 1 is not expected to be fixed in future hardware [46]. We assert that JumpSwitches are as safe as future "Spectre safe" processors currently on vendor roadmaps.

Compared to indirect call promotion implemented with a compiler, JumpSwitches are able to dynamically adapt to changing workloads and take advantage of semantic information only available at runtime, much as a JIT compiler may employ polymorphic inline caching [22]. Unlike binary translation, JumpSwitches are integrated in the kernel and designed for minimal overhead to only instrument indirect calls rather than the entire binary. Furthermore, JumpSwitches are able to take advantage of the rich semantic information available in the kernel source and lost in the binary. A summary of indirect call mechanisms can be found in Table 1.

Our work is focused on the Linux kernel since it fully supports retpolines and runs privileged code which must be protected against Spectre. We believe, however, that JumpSwitches can be applied as a general mechanism and be widely deployed as retpolines are today. The goals of JumpSwitches are to:

- Predict indirect branch targets while preserving safety.
- Require minimal programmer effort: leveraging JumpSwitches should not require source code changes.
- Be flexible: allow the programmer to provide additional semantic information when available.

3 JumpSwitch Architecture

JumpSwitches are code fragments which, like retpolines, serve as trampolines for indirect calls. Their purpose is to leverage indirect call promotion and use direct calls which have a much lower cost than indirect calls, especially in the era of Spectre. JumpSwitches are Spectre aware: if a JumpSwitch cannot promote an indirect call, a Spectre mitigated indirect call is made. In Spectre-vulnerable hardware, JumpSwitches fall back to retpolines, but future hardware may easily take advantage of technologies such as Enhanced IBRS.

We have developed five different types of JumpSwitches, each optimized for a different purpose. The simplest and default type of JumpSwitch is known as an *Inline JumpSwitch*, which is optimized for code size and covers the majority of use cases. The *Outline JumpSwitch* is used when we learn that an indirect branch has multiple targets. Inline and outline JumpSwitches are known as *generic JumpSwitches* and can be used on all indirect branches. We also provide three *semantic JumpSwitches* which support commonly encountered indirect branches where deeper semantic information is available from the programmer. *Search JumpSwitches* support a large number (hundreds) of targets. The *Registration JumpSwitch* covers the commonly used registration pattern used in callback lists. Finally, the *Instance JumpSwitch* covers the case where call targets are strongly correlated with an instance, such as an object or a process. Within the Linux kernel, the *JumpSwitch worker* learns about new branch targets and makes decisions on whether a JumpSwitch should be changed to a different type, outside the path of execution. To change the active JumpSwitch, the worker makes use of a multi-stage patching mechanism which atomically updates a JumpSwitch without risking safety.

JumpSwitches are integrated into the Linux build infrastructure through the use of a compiler plugin. Thanks to our integration with the Linux build system, taking advantage of inline JumpSwitches and outline JumpSwitches requires no source code changes. We also supply additional JumpSwitches which leverage semantic information available in the kernel provided by developers, enabling the search, registration and instance JumpSwitches.

In the next sections, we first describe each type of JumpSwitch. Then, we show how the JumpSwitch worker learns and adapts JumpSwitches during runtime. Finally, we show how we patch Linux with JumpSwitches as well as optimizations we made during integration.

3.1 JumpSwitch Types

Generic JumpSwitches can be used on all indirect calls, while semantic JumpSwitches cover common use cases. We enumerate the types of JumpSwitches below:

Inline JumpSwitch. These JumpSwitches serve as generic trampolines which replace an indirect call. They act as a basic building block for other JumpSwitches and enable dynamic learning of branch targets. Because they replace code, they must be short, otherwise they risk bloating code and increasing pressure on the instruction cache. At the same time, they must be upgradable by the JumpSwitch worker at runtime and support learning. In order to fulfill these three requirements, inline JumpSwitches are designed to be safe by default and easily patched. An example of an inline JumpSwitch and the indirect call it replaces is given in Figure 3.

```
1   call %eax        cmp ${likely},%eax
2                    jnz miss
3                    call {likely rel}
4                    jmp done
5              miss: call {slow rel}
6              done:
```

Figure 3: Inline JumpSwitch. Indirect branches on the left are replaced with a sequence that may promote a *likely* call to a direct call (lines 1–4), falling back to a *slow* path if the likely target was not in **%eax**.

```
1 cmp    ${ entry0 }, %eax
2 jz     { entry0 rel }
3 cmp    ${ entry1 }, %eax
4 jz     { entry1 rel }
5        ...
6 jmp    learning rel
```

Figure 4: Outline JumpSwitch. Multiple targets are supported, using `JZ` instructions to avoid the need for a function epilogue and falling back to learning if the target is not found.

The inline JumpSwitch may point to one of two targets: *likely* or *slow*. *Likely* represents a branch target that the JumpSwitch has learned to be likely and is promoted to avoid an indirect jump. *Slow* can represent one of three targets, depending on which mode the inline JumpSwitch is in:

- Learning mode, where *slow* points to learning code which updates a table of learned targets.
- Outline mode, where *slow* points to an outline JumpSwitch leading to more targets.
- Fallback mode, where *slow* points to either a retpoline or is a normal indirect call, depending on if the system is Spectre vulnerable.

When compiled, the inline JumpSwitch is set to fallback mode and both *likely* and *slow* point to a retpoline. At runtime, the JumpSwitch worker may patch *likely* and *slow* depending on which mode is required and what targets have been learned. When an inline JumpSwitch is executed, if *likely* matches the contents of the register holding the branch target (in this case, *eax*), the call on line 3 is executed. Otherwise, the call to *slow* is executed. It is important to note that while the value of *likely* is used by the `CMP` instruction represents a direct address, the target of the `CALL`s are relative, since there are no direct absolute jumps in x86-64.

Outline JumpSwitch. To support multiple targets dynamically without increasing code size in the common case where there is only a single learned target, outline JumpSwitches may be called by inline JumpSwitches. Unlike inline JumpSwitches, outline JumpSwitches are dynamically allocated and generated by the JumpSwitch worker as targets are learned. As an optimization, since outline JumpSwitches are called by inline JumpSwitches, we avoid the normal work of setting the frame pointer and returning by using jump instructions, as shown in Figure 4. As a result, each target uses two instructions: a `CMP` and a `JZ`. If the target is not in the outline JumpSwitch, we fall back to learning code, as in the inline JumpSwitch in learning mode.

While outline JumpSwitches support multiple targets, each target adds an additional conditional branch, which induces overhead. To avoid reducing performance due to excessive or unpredictable branch targets, we limit targets in an outline JumpSwitch to 6, for a total of 7 (1 in the inline JumpSwitch and 6 in the outline). To support more targets, a search JumpSwitch can be used.

In addition to the generic indirect branches targeted by inline and outline JumpSwitches, we also target common indirect branches when semantic information is available.

Registration JumpSwitch. The registration pattern occurs when a list of callbacks is registered for later use. Registration is used in the kernel for structures such as callback lists (e.g., notifiers such as `user_return_notifier` and `mmu_notifier`), or filter lists (e.g. `seccomp`).

In such cases, since the callbacks are called from a single call-site in a loop, learning targets would fail if the callback list length is greater than the maximum number of learning JumpSwitch targets. Instead, we use a registration JumpSwitch, which unrolls callback list invocation code, sets multiple call instructions to callbacks. When a function is added or removed callbacks from a callback list the registration JumpSwitch is explicitly invoked and patches the callback addresses into the call instructions.

Instance JumpSwitch. Another common pattern that JumpSwitches target are cases where the likely branch targets are strongly correlated with a process. For example, the running process may dictate which `seccomp` filters are running, or per-process `preemption_notifier` used. Instance JumpSwitches take advantage of semantic knowledge about the running workload, and contain one of the previous three JumpSwitch types, but on a per-instance basis.

To support instance JumpSwitches, a separate executable memory area is allocated for each process. This memory area contains the instance JumpSwitches. While each area is located in a different physical memory address, the instance JumpSwitch is always mapped in a fixed virtual address. This allows us to invoke process specific JumpSwitches by direct calls, as context switches between different processes also switch instance JumpSwitches to the process-specific ones. Learning and code modifications are then done on a per-process basis.

```
1  int syscall(int nr, regs_t *regs)
2  {
3    int direct_nr = private_nr[nr];
4
5    if (direct_nr == INVALID)
6          return call_table[nr](regs);
7
8    /* Hit; 4 entries per-process */
9    if (direct_nr < 2) {
10     if (direct_nr < 1)
11       return private_call_0(regs);
12     return private_call_1(regs);
13   }
14   if (direct_nr < 3)
15     return private_call_2(regs);
16   return private_call_3(regs);
17 }
```

Figure 5: Search JumpSwitch pseudo-code, similar to the one that is used to invoke system-calls. In this example there are 4 slots for the most common targets. If there is a miss, an indirect call is initiated (line 6). Otherwise, a direct branch is performed (lines 9-16). System-call indirection table and functions (prefixed with "private") are set per instance (i.e., process). They are located at a fixed virtual address, by mapped to different physical memory on each process.

Search JumpSwitch. Some indirect branches may have potentially hundreds of targets, such as in call tables (e.g., syscall, hypercalls) and event handlers (e.g., virtualization traps) and other jump tables commonly used to execute selection control, as commonly done by C "switch" statements. These constructs typically translate a key such a handler number to a function for that key, and can be compiled to machine code that use binary decision tree or to code that use jump-tables. In practice, compilers prefer jump-tables for densely packed case items, as they usually require fewer instructions and branches [8]. However, this behavior is based on the assumption that an indirect branches are inexpensive, an assumption which has changed and should be reevaluated in the era of Spectre, particularly when retpolines are required.

In these cases, the JumpSwitch may benefit from a search tree which may reduce the number of branches needed for an indirection lookup and support many more targets. However, in the Linux kernel, we experimented with the most commonly used table, the system call table, which is used to dispatch the function that handles system calls based on a well known ("magic") number. Linux implements this table manually and does not use a switch statement. In x86 Linux, there are over 300 system calls. When retpolines are enabled, we found both static jump tables and binary decision trees to be inefficient. Our experiments with static binary decision tree showed they easily degrade performance when more than very few system calls are used. Unfortunately, outline JumpSwitches do not perform well as all the slots quickly fill up and the learning mechanism disables the outline block to prevent excessive overhead and performance degradation.

To address this situation, search JumpSwitches use an adaptive binary decision tree that caches the most frequent call translations (in the case of a system call, from system call number to handler), and the jump-table is used as a fallback. We dynamically construct a binary decision tree and keep a bitmap, where each bit corresponds to the respective translation in the jump-table. When the translation is called, the search JumpSwitch checks and updates the bitmap if learning is enabled, then dispatches the request (Figure 5). The JumpSwitch worker periodically updates the decision tree based on the learned frequency data.

3.2 Learning and the JumpSwitch Worker

To learn new targets, JumpSwitches use a learning routine, which buffers learned targets in an optimized hash table, and the worker periodically updates JumpSwitches using multi-stage patching. Different routines are used for learning on generic (inline and outline JumpSwitches) and search JumpSwitches. Registration and instance JumpSwitches do not learn asynchronously and do not require the worker.

Generic Learning Routine. To learn new branch targets, we could have used hardware mechanisms such as Precise Event-Based Sampling (PEBS) [24], which is used by many PGO and FDO frameworks. However, PEBS has several limitations, such as not being able to selectively profile branches and cannot be run in most virtual environments. To ensure that JumpSwitches are hardware agnostic, we developed a lightweight software mechanism for learning branch targets.

Our software learning routine records branch targets in a small 256-entry per-core hash table. The branch source and destination instruction pointers are XOR'd and the low 8 bits are used as a key for the table. Each entry of the table saves the instruction pointer of the source and the destination (only the low 32-bits, as the top 32-bits are fixed), as well as a counter of number of invocations. We ignore hash collisions, which can potentially cause destinations that do not match the source to be recorded and wrong invocation count, as they only lead to suboptimal decisions in the worst case and do not affect correctness. This allows us to keep the learning routine simple and short (14 assembly instructions), which is important for keeping the overhead of learning low.

After the learning routine is done, fallback code is called, which may either be a retpoline if Spectre-vulnerable hardware is present or a normal indirect call.

Search Learning Routine. For search JumpSwitches, an alternate method is used. Search JumpSwitches are tracked

per process. Each thread holds a flag that indicates whether learning needs to be done. When learning is on, each thread records which calls have been performed in a per-core frequency table. Upon context-switch the frequencies are added to a per-process frequency table that sums them up, and the per-core table is cleared. When training is over, an interprocessor interrupt (IPI) is sent to cores that still run threads of the process to sum them up into the per-process table.

JumpSwitch Worker. The worker runs once every epoch (1 second default, configurable), or when a relearning event is triggered. The worker performs learning by reading the targets that were discovered by the learning routines and updating the JumpSwitch. The worker processes generic JumpSwitches and search JumpSwitches differently:

Generic JumpSwitch Updates. During each epoch, the JumpSwitch worker checks if new call targets were encountered by reading the hash tables on each core and summing the total calls for each source-destination pair on all cores. For each source, the worker sorts each destination by the number of hits and promotes those destinations. If the destinations have already been promoted, they are ignored, and if the maximum capacity of an outline JumpSwitch is reached, the inline JumpSwitch is put into fallback mode, which disables learning for the target. As a result, a worker run can result in the following changes to a JumpSwitch:

- Update an inline JumpSwitch's *likely* target.
- Switch an inline JumpSwitch from learning to outline.
- Create or add targets to an outline JumpSwitch.
- Switch an inline JumpSwitch to fallback mode.

Once the worker is done processing data, it clears all hash tables allowing calls with hash conflicts to save their data.

Generic Learning Policy. Learning imposes some overhead and should only be performed when a performance gain will likely result. To mitigate learning overheads, the worker holds generic JumpSwitches in three lists:

- Learning JumpSwitches, which are in learning mode. They do not improve performance, but track targets.
- Stable JumpSwitches, which have a single target. These JumpSwitches do not need to be disabled for relearning, as their fallback path jumps to the learning routine.
- Unstable JumpSwitches, which have more than a single target. These include JumpSwitches with an outlined block, and those that have too many targets, and were therefore set not to have an outlined block.

During each epoch, if no JumpSwitches were updated, the JumpSwitch worker picks a number of JumpSwitches (configurable, 16 default) from the unstable list and converts them into learning JumpSwitches, disabling them and setting the fallback to jump to the learning routine. To avoid being too aggressive, the worker does not switch a JumpSwitch into learning mode more than once every 60 seconds.

Search Learning Updates. For search JumpSwitches, the worker sums up per-cpu frequency tables, but instead of updating an inline JumpSwitch, a binary-search tree is updated, promoting frequent values and clearing the bitmap.

Search Learning Policy. Learning is turned on periodically per-process. For each process, we save whether learning is on and the last time it was performed. When a thread is scheduled to run, it checks if learning is on. If so, it caches the status in thread-local memory. If it is off, a check is performed to see whether a time interval passed since learning was last on (20 seconds default). If that time has elapsed, learning is turned back on, and the process is added into a list of learning processes. Learning is stopped on the next update.

3.3 Patching and Updating

To minimize the performance impact of patching inline JumpSwitches, the worker employs a multi-phase mechanism to ensure that JumpSwitches are safely updated as multiple instructions are patched live without locks or halting execution. We leverage the Linux `text_poke` infrastructure [29], designed for live patching of code.[2] Patching is a three phase process, outlined below, and line numbers refer to inline JumpSwitch shown on the right of figure 3.

1. A breakpoint is installed on line 1 by writing the single-byte breakpoint opcode onto the first byte of the instruction. If the breakpoint is hit, the handler emulates a call to the retpoline, as if it was executed on line 5. To simplify implementation, the handler does not perform the emulation directly, but instead moves the instruction pointer to a newly created chunk that pushes onto the stack the return address (line 6) and executes `JMP` branch to the retpoline chunk.
2. The patching mechanism waits for a quiescence period, to ensure no thread runs the instructions in lines 2–5. In the Linux kernel, this is performed by calling the `synchronize_sched` function. Afterwards, the instructions on lines 2, 3 and 5 are patched. The instruction on line 4 is not changed, to allow functions that return from the `CALL` in line 3 to succeed in completing the JumpSwitch.
3. The same breakpoint mechanism in phase 1 is used, this time restoring the `CMP` on line 1 with the new promoted target, re-enabling the JumpSwitch.

When a fully-preemptable kernel is used, we also check whether the JumpSwitch code was interrupted before the target function was called and rewind the saved instruction

[2] We have submitted upstream patches to Linux to further harden the security of this mechanism [5].

pointer to line 1. This ensures the code will be executed again when the thread is re-scheduled. To efficiently determine if JumpSwitch code was interrupted, a check is only performed only if instructions we use (cmp, jnz, jz, call and jmp) are interrupted.

3.4 Linux Integration

We implement and integrate JumpSwitches on Linux v4.18 through a gcc-plugin integrated into the Linux build system. The gcc-plugin is built during the kernel build and replaces call sites with our JumpSwitch code, allowing generic JumpSwitches to be seamlessly integrated into the Linux kernel without source code changes. We write the instruction pointer and the register used for the indirect call into a new ELF section. This information is read during boot to compose a list of calls, allowing the worker to easily realize which register is used in each JumpSwitch. It also serves as a security precaution to prevent intentional or malicious memory corruption of the JumpSwitch sampling data from causing the JumpSwitch worker from patching the wrong code. The use of JumpSwitches is configurable via a Kconfig `CONFIG` option.

Semantic JumpSwitches require slight changes to kernel source. To support instance JumpSwitches for processes, we mapped a per-process page in kernel space. We implemented both search and registration JumpSwitches to be placed in per-process instance JumpSwitches. We modified `seccomp` to use a registration JumpSwitch, which accounts for the fact that filters are per process by being placed in an instance JumpSwitch. We also patch system call dispatching to use a search JumpSwitch, to account for the large system call table which is used, and place it in an instance JumpSwitch. Overall, implementing our semantic JumpSwitches in the kernel required changing about 30 SLOC.

The JumpSwitch worker is integrated into the kernel in a similar manner to other periodic tasks in Linux which patch code such as the `static-keys`, `jump-label` and `alternatives` infrastructure in Linux.

3.5 Direct Kernel Entry

During our Linux integration, we observed that JumpSwitches did not provide the full speedup we expected. Further analysis revealed that this was due to the overhead of page-table isolation (PTI), which was introduced to mitigate against Meltdown, a different speculative execution vulnerability [1, 34]. PTI introduced a new trampoline used during system calls to switch between the user and kernel page tables. This trampoline is mapped to a different virtual address on each core so the trampoline can determine the correct per-core transitional kernel stack. The trampoline is part of a per-core data structure, and since this data-structure is large, it is located "far" (more than 2GB away from the kernel [19]), preventing a direct jump. Unfortunately, the resulting indirect jump must use a retpoline [52] on Spectre vulnerable hardware, resulting in lower than expected gains when we used JumpSwitches.

To eliminate the need for this retpoline, we split the per-core data structure into two data structures: a small one which includes the transitional kernel stack, TSS and trampoline code, which are all needed to transition into the kernel during a system call; and a second larger one that includes the other fields. This allowed us to move the small part of the trampoline back into the same 2GB in which the kernel code is mapped, and use a relative jump instead of a retpoline, resulting in a significant performance gain. Our solution requires replicating the trampoline page, which consumes a minimal amount of extra physical memory ($<$ 32MB for 8192 cores).

4 Evaluation

Our evaluation is guided by the following questions:

- How does the specialization of each JumpSwitch impact the performance of the kernel in isolation? (§4.1)
- How do JumpSwitches perform with real-world applications and benchmarks? (§4.2)
- How does learning impact JumpSwitches? (§4.3)
- How many targets are needed per indirect branch? (§4.4)
- Is JumpSwitch is useful after the recent security vulnerabilities are resolved in hardware? (§4.5)

Testbed. Our testbed consists of a Dell PowerEdge R630 server with Intel E5-2670 CPUs, a Seagate ST1200 disk, which runs Ubuntu 18.04. The benchmarks are run on guests with a 2-VCPUs and 16GB of RAM. Each measurement was performed at least 5 times and the average result and standard deviation are reported. All workloads were executed with a warm-up run prior to measurement.

Configurations. We run and report the speedup relative to the baseline system which uses retpolines as a mitigation against Spectre. We report the results of:

- **base**: The baseline system with retpolines enabled.
- **direct-entry**: direct jump kernel entry trampoline.
- **+inline**: *direct-entry* with inline JumpSwitches.
- **+outline**: *+inline* with outline JumpSwitches when there are multiple targets.
- **+registration**: *+outline* with per-process (instance) registration JumpSwitches for `seccomp`
- **+search**: *+registration* with per-process (instance) search JumpSwitches for system calls.
- **unsafe**: the baseline system with retpolines disabled.

4.1 Microbenchmarks

Given the diversity of JumpSwitches we implemented, we first wanted to evaluate how each type of JumpSwitch would

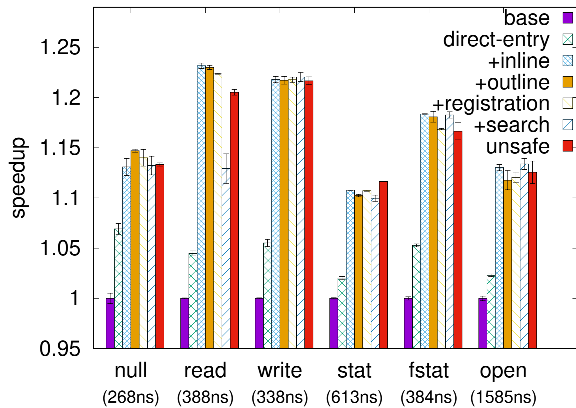


Figure 6: System call speedup relative to the baseline setup that uses retpolines. The runtime of the baseline system is reported in parentheses.

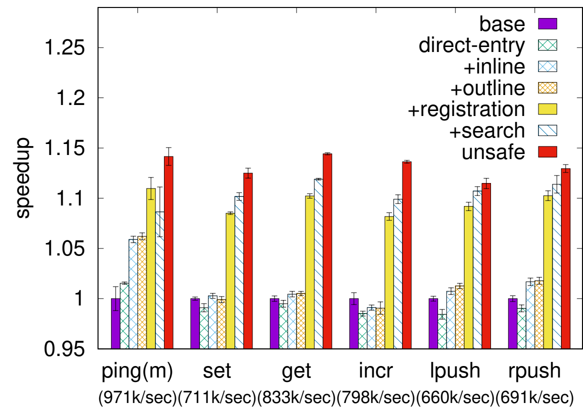


Figure 7: Relative speedup of Redis using JumpSwitch. The operations are are invoked and measured by redis-benchmark. The runtime of the baseline system is reported in parentheses.

improve the performance of the kernel in isolation. We evaluate system calls in a microbenchmark as they have shown to a particular source of slowdown as a result of both the Meltdown [1] and Spectre hardware vulnerabilities, as system calls stress user-kernel transitions, which must now be protected.

We measure the impact of JumpSwitch on the time it takes to run common system-calls using the `lmbench` tools for performance analysis [35]. Figure 6 shows the speedup relative to the baseline protected system. As shown, eliminating the indirect call in the entry trampoline provides a benefit of roughly 15ns which is more pronounced in short system calls. The inline JumpSwitch improves performance by up to 15%, making the system calls run as fast as they would without retpolines. It is noteworthy that this is due to the fact that the workload is very simple, which allows the training mechanism to inline call targets with very high precision.

This high precision is the reason that Outline JumpSwitches do not provide any benefit in this benchmark. Semantic JumpSwitches also offer little benefit here: `seccomp` filters are not installed and the same system call is called repeatedly.

To further understand the performance benefits of JumpSwitches, we some simple operations: Redis key-value store commands using redis-benchmark. Snapshotting is disabled to reduce variance. Each test runs the same command repeatedly, and the results are depicted in Figure 7. As seen, using registration JumpSwitches to avoid indirect calls when `seccomp` filters are invoked, provide up to 9%, performance improvement. This is due to the fact that `systemd`, software which acts as a system and service manager in Ubuntu, attaches 17 `seccomp` filters, which are executed upon each system call. Search JumpSwitches, whose benefit was not shown when the same system call was repeatedly executed, improve performance by up roughly 2%.

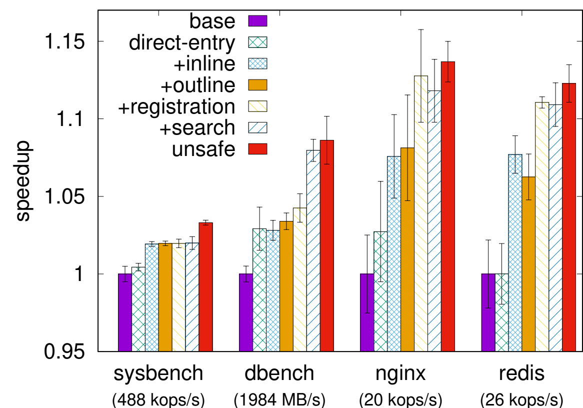


Figure 8: Relative speedup of macro-benchmarks using JumpSwitch. The baseline system is reported in parentheses.

4.2 Macrobenchmarks

Next, we want to see how JumpSwitches perform with real-world workloads which do not necessarily stress the userspace-kernel transition. We run the following benchmarks: *sysbench*, which runs a mixture of file read, write and flush operations, running on a temporary file-system (tmpfs); *dbench*, a disk benchmark that simulates file server workload, running on tmpfs; Nginx web-server, using ApacheBench workload generator to send 500k requests for a static webpage using with 10 concurrent requests at a time; and *Redis*, using Yahoo Cloud System Benchmark (YCSB) as a workload generator (running `workloadA`). In all cases we ensure the workload generator is not the bottleneck.

Figure 8 depicts the results. Eliminating the indirect branch from the entry trampoline provides a modest performance

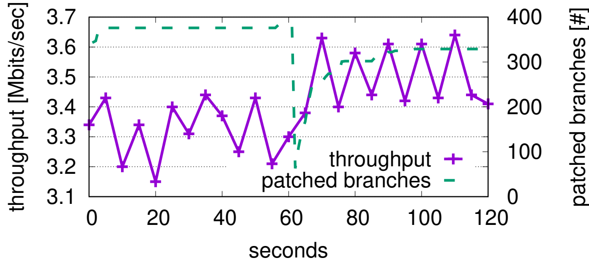


Figure 9: UDP throughput and the number of patched branches when the workload changes and learning is initiated after 60 seconds. A single inlined JumpSwitch is used.

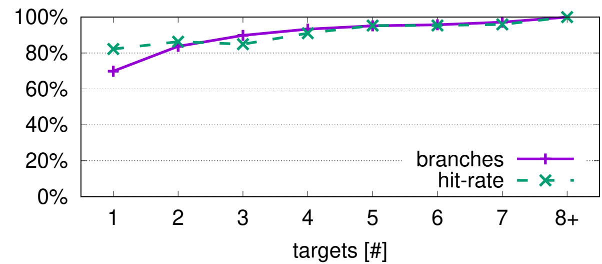


Figure 10: Hit-rate of the inlined and outlined JumpSwitch compared against a CDF of the number of targets that each indirect call-site holds while running Nginx.

improvement, which is most pronounced in dbench as it invokes many system calls. The inline JumpSwitch provides the major part of the performance gains, up to 8% improvement over the baseline system for the Redis benchmark. Like the Redis micro-benchmarks, the macro-benchmark shows a considerable gain (3%) from a registration JumpSwitch, due to the multiple (17) `seccomp` filters which are attached to it. Dbench shows a performance gain of up to 4% from the search JumpSwitch, as it repeatedly runs a small subset of system calls, which are quickly learned by the adaptive search tree. It appears that this mechanism also improves Redis performance, but due to the high standard deviation, it is hard to say so definitively. For the same reason it is hard to conclude whether the experienced performance degradation in nginx with some mechanisms is meaningful, especially since registration JumpSwitches have almost no effect on nginx, whose system calls do not go through `seccomp` filters.

Overall, the macro-benchmarks evaluation show that JumpSwitches can restore most of the performance loss due to retpolines, narrowing the difference between protected and non-protected systems to less than 3%.

4.3 Dynamic Learning

One of the main benefits of the runtime instrumentation of JumpSwitches over compile-time decisions is the ability to dynamically learn branch targets. To evaluate the value, effectiveness and performance of dynamic learning we create a scenario where the workload behavior changes. In this experiment we only enable inline JumpSwitches, emulating how compilers perform indirect branch prediction. To control learning, we disable the automatic learning mechanism and use a user-visible knob that initiates the relearning.

First, we run `iperf`—a network bandwidth measurement tool—to send and receive UDP packets using IPv6, and we set the kernel to learn and adapt the branches accordingly. Then we use iperf to measure the throughput of IPv4 UDP performance, by sending messages of a single byte. After 60 seconds, we restart the learning process.

Figure 9 shows the throughput (sampled every 5 seconds), and the number of adapted branches (sampled every second). As shown, restarting the learning process resets all the branches, dropping the number of patched branches to zero momentarily. Yet, within 5 seconds, over half of the branches that were patched prior to the benchmark execution are patched again with updated targets. The adaptation of the branches improves the overall throughput by ≈5%.

This demonstrates the value of dynamic learning over profile-based techniques, which "bake" in indirect branch promotions at compile time. Since learning is fast, it can be done periodically without degrading the overall performance of the system. Future work may apply a more advanced algorithm to do relearning, such as on a system event.

4.4 Branch Targets

The usefulness of inline and outline JumpSwitches depends on the number of branch targets and the distribution of the frequency in which they are used. To study this distribution, we used our system by modifying the number of branch target slots in each JumpSwitch and measuring the hit rate. In addition, we measured how many targets each branch has and created a cumulative distribution function to compare with the hit-rate. As shown in Figure 10, 71% of the function calls had a single target, and the inlined JumpSwitch by itself (only 1 allowed destination) achieved a hit-rate of 82%. As we increase the number of allowed destinations, the outline JumpSwitch further improves the hit-rate up to 96% when both outlined and inlined JumpSwitch are used.

This shows that inline JumpSwitches alone are able to handle a majority of cases, while outline JumpSwitches provide more complete coverage for branches with many targets. Only few branches require beyond 7 targets. In the small fraction of branches with 8 or more targets, outline JumpSwitches are disabled because the cost of the additional conditional branches outweigh the benefits of call promotion.

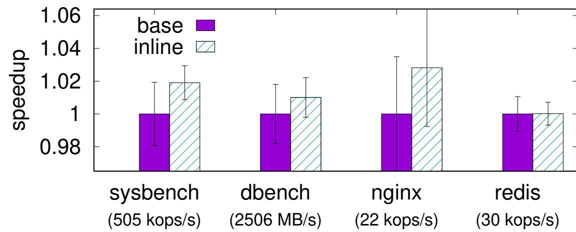


Figure 11: Benchmark speedup relative to the baseline when Spectre and Meltdown protections are disabled. The runtime of the baseline system is reported in parentheses.

4.5 Post-Spectre Benefits

Finally, we examine whether JumpSwitches are relevant when hardware solutions for Spectre and Meltdown mitigations are present, and retpolines are not longer needed. The benefit of JumpSwitches would be smaller, since the cost of an indirect branch becomes considerably lower. Some of our proposed mechanisms will become irrelevant. If Meltdown is resolved, the indirect branch in the trampoline page is not used. If Spectre is resolved, search and outline JumpSwitches become too costly relatively to the cost of an indirect branches. Registration JumpSwitches may improve performance, but our experiments indicate they require further micro-optimizations.

In contrast, inline JumpSwitches are potentially valuable even after the recent CPU bugs are fixed. Their performance benefit might be lower, making aggressive retraining of inline JumpSwitches inappropriate, yet retraining can still be done infrequently, based on user requests or in response to system events (e.g., CPU saturation). To evaluate the impact of JumpSwitches in such setups, we run the same macro-benchmarks while directing Linux to drop the protections against Meltdown (no page table isolation) and Spectre (no retpoline). We disable the automatic relearning mechanism and relearn manually when the workload is first invoked.

The results are shown in Figure 11. JumpSwitch can provide up to 2% performance improvement on systems which that are not vulnerable to Spectre and Meltdown. As shown on Redis, however, this benefit can be nullified in some cases.

5 Conclusion

The recent CPU vulnerabilities due to speculative execution revealed that the CPU cannot be regarded as a black-box that can be blindly relied on. Hardware bugs are hard to fix in existing systems, which necessitates the mitigation against the vulnerabilities using software techniques. JumpSwitch performs this task by extending compiler-optimization to make runtime decisions, while reducing the overhead of the current mitigation techniques. We have shown that JumpSwitches achieve our goal of leveraging speculative execution cycles to predict indirect branch targets while preserving safety, requiring minimal programmer effort while providing the flexibility for the programmer to add rich semantic information.

JumpSwitches show that software can efficiently perform hardware tasks such as branch prediction. Since proposed hardware mitigations against speculation will come with a cost in performance, a hardware-software solution would allow software to define in fine granularity which speculation is permitted and which needs to be blocked. JumpSwitches limit the allowed speculation using direct branches. Hardware mechanisms provide tools for software to perform this task more efficiently, for example, by providing raw interfaces to content addressable-memory. The benefit of combining both solutions is both in performance, by leveraging software knowledge, and in security, by allowing easier mitigation of potential hardware vulnerabilities.

6 Acknowledgment

We would like to thank the paper shepherd, Yuval Yarom and the anonymous reviewers for the insightful comments. We would also like to thank Andy Lutomirski, Josh Poimboeuf and Peter Zijlstra for their imperative technical feedback. This research was supported by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, grant 0605714731.

References

[1] CVE-2017-5754. Available from NVD, CVE-ID CVE-2017-5754, https://nvd.nist.gov/vuln/detail/CVE-2017-5754, January 1 2018. [Online; accessed 21-May-2019].

[2] Paolo Abeni. Patch: net: mitigate retpoline overhead. Linux Kernel Mailing List, https://lwn.net/ml/linux-kernel/cover.1544032300.git.pabeni@redhat.com/, 2018. [Online; accessed 21-May-2019].

[3] Varun Agrawal, Amit Arya, Michael Ferdman, and Donald Porter. Jit kernels: An idea whose time has (just) come. http://compas.cs.stonybrook.edu/~mferdman/downloads.php/SOSP13_JIT_Kernels_Poster.pdf. [Online; accessed 21-May-2019].

[4] Nadav Amit. Rfc dynamic indirect call promotion. Linux Kernel Mailing List, https://lkml.org/lkml/2018/10/18/175, 2018. [Online; accessed 21-May-2019].

[5] Nadav Amit. PATCH: x86: text_poke() fixes and executable lockdowns. https://lore.kernel.org/patchwork/cover/1067359/, 2019. [Online; accessed 21-May-2019].

[6] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *ACM SIGPLAN Notices*, volume 32, pages 134–145. ACM, 1997.

[7] Ivan Baev. Profile-based indirect call promotion. 2015. [Online; accessed 21-May-2019].

[8] Robert L Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985.

[9] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408. ACM, 1994.

[10] 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. ACM, 2016.

[11] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013.

[12] Jonathan Corbet. Finding spectre vulnerabilities with smatch. Linux Kernel Mailing List, `https://lwn.net/Articles/752408/`, 2018. [Online; accessed 21-May-2019].

[13] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. 6 2016. [Online; accessed 21-May-2019].

[14] Intel Corporation. Intel analysis of speculative execution side channels. `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`, 1 2018. [Online; accessed 21-May-2019].

[15] Intel Corporation. Retpoline: A branch target injection mitigation - white paper. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`, 2018. [Online; accessed 21-May-2019].

[16] Intel Corporation. Speculative execution side channel mitigations. 5 2018. [Online; accessed 21-May-2019].

[17] Matthew Dillon. Clarifying the Spectre mitigations... `http://lists.dragonflybsd.org/pipermail/users/2018-January/335637.html`, 2018. [Online; accessed 21-May-2019].

[18] Timothy Garnett. *Dynamic optimization if IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, 2003.

[19] Thomas Gleixner. x86/cpu_entry_area: Move it out of fixmap. Linux Kernel Mailing List, `https://lore.kernel.org/patchwork/patch/866046/`, 2017. [Online; accessed 21-May-2019].

[20] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 68–78. IEEE Computer Society, 2015.

[21] Christoph Hellwig. RFC: avoid indirect calls for DMA direct mappings v2. Linux Kernel Mailing List, `https://lwn.net/ml/linux-kernel/20181207190720.18517-1-hch@lst.de/`, 2018. [Online; accessed 21-May-2019].

[22] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[23] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM SIGPLAN Notices*, volume 29, pages 326–336. ACM, 1994.

[24] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2016.

[25] Jose A Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 42, pages 80–90, 2008.

[26] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 111–121. IEEE, 2017.

[27] 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. ACM, 2013.

[28] Hyesoon Kim, José A Joao, Onur Mutlu, Chang Joo Lee, Yale N Patt, and Robert Cohn. Vpc prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ACM SIGARCH Computer Architecture News (CAN)*, volume 35, pages 424–435, 2007.

[29] Andi Kleen. Add a text_poke syscall. LWN.net https://lwn.net/Articles/574309/, 2013. [Online; accessed 21-May-2019].

[30] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[31] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *arXiv preprint arXiv:1807.07940*, 2018.

[32] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61. ACM, 2010.

[33] Tao Li, Ravi Bhargava, and Lizy Kurian John. Adapting branch-target buffer to improve the target predictability of Java code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(2):109–130, 2005.

[34] 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)*, 2018.

[35] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[36] Microsoft. Mitigating speculative execution side channel hardware vulnerabilities. https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018. [Online; accessed 21-May-2019].

[37] MIT Technology Review. At least three billion computer chips have the spectre security hole. https://www.technologyreview.com/s/609891/at-least-3-billion-computer-chips-have-the-spectre-security-hole/, 1 2018. [Online; accessed 21-May-2019].

[38] MITRE. CVE-2017-5715: branch target injection, spectre-v2. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715, 2018. [Online; accessed 21-May-2019].

[39] MITRE. CVE-2017-5753: bounds check bypass, spectre-v1. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753, 2018. [Online; accessed 21-May-2019].

[40] Paweł Moll. FDO: Magic "make my program faster" compilation option? https://elinux.org/images/4/4d/Moll.pdf, 2016. [Online; accessed 21-May-2019].

[41] Sungdo Moon, Xinliang D Li, Robert Hundt, Dhruva R Chakrabarti, Luis A Lozano, Uma Srinivasan, and Shin-Ming Liu. Syzygy-a framework for scalable cross-module ipo. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 65. IEEE Computer Society, 2004.

[42] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. *arXiv preprint arXiv:1807.06735*, 2018.

[43] Josh Poimboeuf. Patch: Static calls. Linux Kernel Mailing List, https://lkml.org/lkml/2018/11/26/951, 2018. [Online; accessed 21-May-2019].

[44] Michiel Ronsse and Koen De Bosschere. JiTI: A robust just in time instrumentation technique. *ACM SIGARCH Computer Architecture News*, 29(1):43–54, 2001.

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307. ACM, 2004.

[46] Ryan Smith. Intel publishes spectre & meltdown hardware plans: Fixed gear later this year. https://www.anandtech.com/show/12533/intel-spectre-meltdown, 2018. [Online; accessed 21-May-2019].

[47] Linus Torvalds. Create macros to restrict/unrestrict indirect branch speculation. 2018. [Online; accessed 21-May-2019].

[48] Linus Torvalds. x86/speculation: Add basic IBRS support infrastructure. 2018. [Online; accessed 21-May-2019].

[49] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. [Online; accessed 21-May-2019].

[50] Vertica. Update: Vertica test results with microcode patches for the meltdown and spectre security flaws. 2018. [Online; accessed 21-May-2019].

[51] David Woodhouse. Fill RSB on context switch for affected cpus. https://lkml.org/lkml/2018/1/12/552, 2018. [Online; accessed 21-May-2019].

[52] David Woodhouse. x86/retpoline/entry: Convert entry assembler indirect jumps. Linux Kernel Mailing List, https://lore.kernel.org/patchwork/patch/876057/, 2018. [Online; accessed 21-May-2019].

[53] David Woodhouse. x86/speculation: Add basic IBRS support infrastructure. 2018. [Online; accessed 21-May-2019].

[54] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in profile-guided operating system kernel optimization. In *ACM Asia-Pacific Workshop on Systems (APSys)*, page 4, 2014.

[55] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Rethinking compiler optimizations for the Linux kernel: An explorative study. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 2. ACM, 2015.

Multi-Queue Fair Queueing

Mohammad Hedayati
University of Rochester

Kai Shen
Google

Michael L. Scott
University of Rochester

Mike Marty
Google

Abstract

Modern high-speed devices (e.g., network adapters, storage, accelerators) use new host interfaces, which expose multiple software queues directly to the device. These *multi-queue* interfaces allow mutually distrusting applications to access the device without any cross-core interaction, enabling throughput in the order of millions of IOP/s on multicore systems. Unfortunately, while independent device access is scalable, it also introduces a new problem: unfairness. Mechanisms that were used to provide fairness for older devices are no longer tenable in the wake of multi-queue design, and straightforward attempts to re-introduce it would require cross-core synchronization that undermines the scalability for which multiple queues were designed.

To address these challenges, we present Multi-Queue Fair Queueing (MQFQ), the first fair, work-conserving scheduler suitable for multi-queue systems. Specifically, we (1) reformulate a classical fair queueing algorithm to accommodate multi-queue designs, and (2) describe a scalable implementation that bounds potential unfairness while minimizing synchronization overhead. Our implementation of MQFQ in Linux 4.15 demonstrates both fairness and high throughput. Evaluation with an NVMe over RDMA fabric (NVMf) device shows that MQFQ can reach up to 3.1 Million IOP/s on a single machine—20× higher than the state-of-the-art Linux Budget Fair Queueing. Compared to a system with no fairness, MQFQ reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload (2× is considered "fair" slowdown).

1 Introduction

Recent years have seen the proliferation of very fast devices for I/O, networking, and computing acceleration. Commodity solid-state disks (e.g., Intel Optane DC P4800X [22] or Samsung PM1725a [38]) can perform at or near a million I/O operations per second. System-area networks (e.g., InfiniBand) can sustain several million remote operations per second over a single link [25]. RDMA delivers data across fabric within a few microseconds. GPUs and machine learning accelerators may offload computations that run just a few microseconds at a time [30]. At the same time, the proliferation of multicore processors has necessitated architectures tuned for independent I/O across multiple hardware threads [4, 36].

These technological changes have shifted performance bottlenecks from hardware resources to the software stacks that manage them. In response, it is now common to adopt a *multi-queue* architecture in which each hardware thread owns a dedicated I/O queue, directly exposed to the device, giving it an independent path over which to send and receive requests. Examples of this architecture include multi-queue SSDs [22, 38, 50] and NICs [42], and software like the Windows and Linux NVMe drivers, the Linux multi-queue block layer [5], SCSI multi-queue support [8], and data-plane OSes [4, 36]. A recent study [51] demonstrated up to 8× performance improvement for YCSB-on-Cassandra, using multi-queue NVMe instead of single-queue SATA.

To support the full bandwidth of modern devices, multi-queue I/O systems are designed to incur no cache-coherence traffic in the common case when sending and receiving requests. It's easy to see why: a device supporting millions of IOP/s sees each new request in a fraction of a microsecond—a time interval that allows for fewer than 10 cross-core cache coherence misses, and is comparable to the latency of a single inter-processor interrupt (IPI). Serializing requests at such high speeds is infeasible now, and will only become more so as device speeds continue to increase while single-core performance stays relatively flat. As a result, designers have concluded that conventional fair-share I/O schedulers, including fair queueing approaches [35, 40], which reorder requests in a single queue, are unsuited for modern fast devices.

Unfortunately, by cutting out the OS resource scheduler, direct multi-queue device access undermines the OS's traditional responsibility for fairness and performance isolation. While I/O devices (e.g., SSD firmware, NICs) may multiplex hardware queues, their support for fairness is hampered by their inability to reason in terms of system-level policies for resource principals (applications, virtual machines, or Linux

cgroups), or to manage an arbitrary number of flows. As a result, device-level scheduling tends to cycle naively among I/O queues in a round robin fashion [44]. Given such simple scheduling, a greedy application or virtual machine may gain unfair advantage by issuing I/O operations from many CPUs (so it can obtain resource shares from many queues). It may also gain advantage by "batching" its work into larger requests (so more of its work gets done in each round-robin turn). Even worse, a malicious application may launch a denial-of-service attack by submitting a large number of artificially created expensive requests (e.g., very large SSD writes) through many or all command queues.

As a separate issue, it is common for modern SSDs [9, 44] and accelerators [20, 32] to support parallel requests internally. Traditional resource scheduling algorithms, which assume underlying serial operation, are unsuitable for devices with a high degree of internal parallelism.

To overcome these problems, we present *Multi-Queue Fair Queueing (MQFQ)*—the first fair scheduler, to the best of our knowledge, capable of accommodating multi-queue devices with internal parallelism in a scalable fashion. As in classical *fair queueing* [13, 34], we ensure that each *flow* (e.g., an application, virtual machine, or Linux cgroup) receives its share of bandwidth. While classical fair queueing employs a single serializing request queue, we adapt the fair queueing principle to *multi-queue* systems, by efficiently tracking global resource utilization and arranging to *throttle* any queue that has exceeded its share by some bounded amount.

Accordingly, we introduce a *throttling threshold* T such that each core can dispatch, without coordinating with other cores, as long as the lead request in its queue is within T of the utilization of the slowest active queue, system-wide. This threshold creates a window within which request dispatches can *commute* [10], enabling scalable dispatch. We show mathematically that this relaxation has a bounded impact on fairness. When $T = 0$, the guarantees match those of classical fair queueing.

The principal obstacle to scalability in MQFQ is the need for cross-queue synchronization to track global resource utilization.We demonstrate that it is possible, by choosing appropriate data structures, to sustain millions of IOP/s while guaranteeing fairness. The key to our design is to *localize* synchronization (intra-core rather than inter-core; intra-socket rather than inter-socket) as much as possible. An instance of the *mindicator* of Liu et al. [29] allows us to track flows' shares without a global cache miss on every I/O request. A novel data structure we call the *token tree* allows us to track available internal device parallelism: an I/O completion frees up a slot that is preferentially reused by the local queue if possible; otherwise, our token tree allows fast reallocation to a nearby queue. Finally, a nonblocking variant of a timer wheel [43, 47] keeps track of queues whose head requests are too far ahead of the shares of their contributing flows: when resource utilization has advanced sufficiently, update of a single index suffices to turn the wheel and unblock the appropriate flows. MQFQ demonstrates that while scalable multi-queue I/O precludes serialization, it can tolerate infrequent, physically localized synchronization, allowing us to achieve both fairness and high performance.

Summarizing contributions:

- We present Multi-Queue Fair Queueing—to the best of our knowledge, the first scalable, fair scheduler for multi-queue devices.
- We demonstrate mathematically that adapting the fair queueing principle to multi-queue devices results in a bounded impact on fairness.
- We introduce the *token tree*, a novel data structure that tracks available dispatch slots in a multi-queue device with internal parallelism.
- We present a scalable implementation of MQFQ. Our implementation uses the token tree along with two other scalable data structures to localize synchronization as much as possible.

2 Background and Design

Fair queueing [13, 34] is a class of algorithms to schedule a network, processing, or I/O resource among competing flows. Each flow comprises a sequence of requests or packets arriving at the device. Each request has an associated cost, which reflects its resource usage (e.g., service time or bandwidth). Fair queueing then allocates resources in proportion to weights assigned to the competing flows.

A flow is said to be *active* if it has any requests in the system (either waiting to be dispatched to the device, or waiting to be completed *in* the device), and *backlogged* if it is active and has at least one outstanding request to be dispatched. Fair queueing algorithms are *work-conserving*: they schedule requests to consume surplus resources in proportion to the weights of the active flows. A flow whose requests arrive too slowly may become inactive and forfeit the unconsumed portion of its share.

Start-time Fair Queueing (SFQ) [18, 19] assigns a start and finish tag to each request when it arrives, and dispatches requests in increasing order of start tags; ties are broken arbitrarily. The tag values represent the point in the history of resource usage at which each request should start and complete according to a system notion of *virtual "time."* Virtual time always advances monotonically and is identical to real time if: (1) all flows are backlogged, (2) the device (server) completes work at a fixed ideal rate, (3) request costs are an accurate measure of service time, and (4) the weights sum to the service capacity. The start tag for a request is set to be the maximum of the virtual time at arrival and the last finish tag of the flow. The finish tag for a request is its start tag plus its cost, normalized to the weight of the flow.

When the server is busy, virtual time is defined to be equal to the start tag of the request in service, and when it is idle, maximum finish tag of any request that has been serviced by

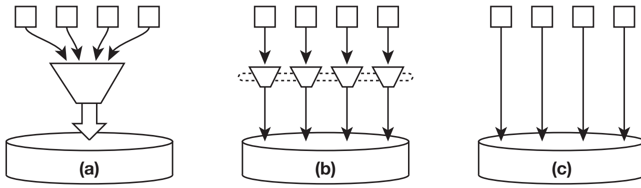


Figure 1: MQFQ (b) employs a set of per-CPU priority queues, rather than (a) a single central queue or (c) fully independent access. Queues coordinate through scalable data structures (suggested by the dotted line; described in Sec. 3) to maintain fairness.

that time. Note that this definition assumes at most a single request can be in service at any moment.

Parallel Dispatch A server with internal parallelism may service multiple requests simultaneously, so virtual time as defined in SFQ is not well-defined in this setting. Moreover, even an active flow may lag behind in resource utilization if it generates an insufficient number of concurrent requests to consume its assigned share.

SFQ(D) [23] works the same as SFQ but allows up to D in-service requests ($D = 1$ reduces to SFQ). Due to out-of-order completion, for the busy server case, virtual time is redefined to be the start tag of the last dispatched request. Note that this definition requires requests to be dispatched in increasing order of start tags, which precludes scalable implementation on multi-queue systems.

2.1 Multi-Queue Fair Queueing

The main obstacle in adapting fair queueing—or most other scheduling algorithms, for that matter—to a multi-queue I/O architecture is the need to dispatch requests in an order enforced by a central priority queue. Additional challenges include the need to dispatch multiple requests concurrently (to saturate an internally parallel device) and the inability to simply advance virtual "time" on dispatch or completion events, since these may occur out of order.

In MQFQ, we replace the traditional central priority queue (Fig. 1(a)) with a set of per-CPU priority queues (Fig. 1(b)), each of which serves to order local requests. To limit imbalance across queues, we suspend (*throttle*) any queue whose lead request is ahead of the slowest backlogged flow in the system (the one that determines the virtual time) by more than some predefined threshold T, allowing other queues to catch up. Setting $T = 0$, while limits scalability in practice, would effectively restore the semantics of a global priority queue. Setting $T > 0$ leads to relaxed semantics but lower synchronization overhead by utilizing the *Scalable Commutativity Rule* [10] to allow requests dispatches to be reordered, i.e., to commute. Specifically, it allows for windows of *conflict free* operations (i.e., no core writes a cache line that was read or written by another core) enabling scalable implementation. While short-term fluctuations of as much as T in the relative progress of flows is possible, it still preserves long-term shares. By adjusting T appropriately, we can find a design point that provides most of the fairness of traditional fair queueing with most of the performance of fully independent queues.

For an internally parallel device, in order to keep the device busy, we will often need to dispatch a new request before the previous one has finished. At the same time, since the device decides the order in which dispatched requests are served, we must generally avoid dispatching more requests than can actually be handled in parallel, thereby preserving our ability to order them. We therefore introduce a second parameter, D, which represents the maximum number of outstanding dispatched requests across all queues.

Recall that a *backlogged* flow is one that has requests ready to be dispatched, and an *active* flow is one that is either backlogged or has requests pending in the device. For any device that supports $D \geq 2$ concurrent requests, the distinction between backlogged and active is quite important: it is no longer the case that an active flow is using at least its fair share (i.e., the flow could be *non-saturating*). In a traditional fair queueing system, an active flow determines the progression of virtual time. With a parallel device, this convention would allow a non-saturating active flow to keep idle resources from being allotted to other flows, leading to underutilization. To fix this, a scheduler aware of internal parallelism needs to use backlogged (instead of active) flows to determine virtual time. We therefore define virtual time (and thus the start tag of a newly arriving request on a previously non-backlogged flow) to be the minimum start tag of all requests across all queues. In a multi-queue system, computing this global minimum without frequent cache misses is challenging. In Sec. 3.1 we show how we localize the misses using a mindicator [29].

The lack of a central priority queue, and our use of the throttling threshold T, raises the possibility not only that requests will complete out of order, but that they may be *dispatched* out of order.

We now define our notion of per-flow virtual time, in a way that accommodates the internal parallelism of the device while retaining the essential property that a *lagging* flow (i.e., a flow that is not backlogged) can never accumulate resources for future use. Recall that queues hold requests that have been submitted but not yet dispatched to the device. The flows that submitted these requests are backlogged by definition. For each such flow f, its virtual time is defined to be the start tag of f's first (oldest) backlogged (waiting to be dispatched) request. (Note that f may have backlogged requests in multiple queues.) Assuming f has multiple pending requests, dispatching this first request would increase f's virtual time by l/r, where r is f's weight (its allotted share of the device) and l is the length (size) of the request. (For certain devices we may also scale the "size" in accordance with operation type—e.g., to reflect the fact that writes are more expensive than reads on an SSD.)

We define global virtual time to be the minimum of per-flow virtual times across all backlogged flows. This is the same

as the minimum of the start tags of the lead requests across all queues, since requests in each queue are sorted by start tags. This equivalence allows us to ignore the maintenance of per-flow virtual times; instead, we directly maintain the global virtual time (hereafter, simply "virtual time") as the minimum start tag of the lead requests across all queues.

As soon as a flow becomes lagging, it stops contributing to the virtual time, which may advance irrespective of a lack of activity in the lagging flow. Request start tags from a lagging flow are still subject to the lower bound of current virtual time. MQFQ then ensures that no request is dispatched if its start tag exceeds the virtual time by more than T. To throttle a flow f that has advanced too far, it suffices to throttle any queues headed by f's requests: since requests in each queue are sorted by start tags, all other requests in such a queue are also guaranteed to be more than T ahead of virtual time.

High-level pseudocode for MQFQ appears in Fig. 2.

2.2 Fairness Analysis

If flows have equal weight, allocation of the device is fair if equal bandwidth is allocated to each (backlogged) flow in every time interval. With unequal weights, each backlogged flow should receive bandwidth proportional to its weight.

If we represent the weight of flow f as r_f and the service (in bytes) that it receives in the interval $[t_1,t_2]$ as $W_f(t_1,t_2)$, then an allocation is fair if for every time interval $[t_1,t_2]$, for every two backlogged flows f and m, we have:

$$\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$$

Clearly, this is possible only if the flows can be broken into infinitesimal units. For a packet- or block-based resource we want

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq H(f,m)$$

to be as close to 0 as possible. $H(f,m)$ is a function of the maximum request lengths, l_f^{max} and l_m^{max}, of flows f and m. Golestani [17] derives a lower bound on the fairness of any scheduler with single dispatch:

$$H(f,m) \geq \frac{1}{2}\left(\frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}\right)$$

We similarly derive bounds on the fairness achieved by MQFQ. Our analysis builds on the fairness bounds for Start-time Fair Queueing (SFQ) [18] and SFQ(D) [23]. Goyal et al. [18] have previously shown in SFQ that in any interval for which flows f and m are backlogged during the entire interval, the difference of weighted services received by two flows at an SFQ server, given as:

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

is twice the lower bound. SFQ uses a single priority queue and serves one request at a time. Now consider an otherwise

```
global structures:
   VT mindicator
   wheel of throttled queues
   token tree of available slots
   set of ready queues (nonempty, unthrottled)

per-flow structures:
   end tag of last submitted request

per-CPU structures:
    local queue of not-yet-dispatched requests

on submission of request R:
   set R's start tag = MAX(VT, per-flow end tag)
   set R's end tag =
      R's start tag + R's service time
   update per-flow end tag
   insert R in local queue
   if R goes at the head
      update VT
   dispatch()

dispatch():
   if local queue is in throttling wheel
      remove it from wheel
   if local queue is in ready queues
      remove it from ready queues
   if local queue is empty
      return
   for lead request R from local queue
      if R's start tag is more than T ahead of VT
         add local queue to throttling wheel
         return
      attempt to obtain slot from token tree
      if unsuccessful
         add local queue to set of ready queues
         return
      remove R from local queue
      deliver R to device
      update VT
      if VT has advanced a bucket's worth
         turn the throttling wheel
         unblock any no-longer-throttled queues
            for which slots are readily available
         add the rest to the set of ready queues

on unblock:
   dispatch()

on request completion:
   choose nearest Q in ready queues (could be self)
   return slot to token tree w.r.t. Q
   unblock Q
```

Figure 2: High-level pseudocode for the MQFQ algorithm. Logic to mitigate races has been elided, as have certain optimizations (e.g., to avoid pairs of data structure changes that cancel one another out).

unchanged variant of SFQ in which the single priority queue is replaced by multiple priority queues with throttled dispatch. We service one request at a time, which can come from any of the queues so long as its start tag is less than or equal to the global minimum + T. We call this variant Multi-Queue Fair Queueing with single dispatch—MQFQ(1).

Theorem 1 *For any interval in which flows f and m are backlogged during the entire interval, the difference in weighted services received by two flows at an MQFQ(1) server with throttling threshold T is:*

$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq 2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

We sketch a proof of Theorem 1 as follows.

Lemma 1 (Lower bound of service received by a flow): *If flow f is backlogged throughout the interval $[t_1, t_2]$, then in an MQFQ(1) server with throttling threshold T:*

$$W_f(t_1, t_2) \geq r_f \cdot (v_2 - T - v_1) - l_f^{max}$$

where v_1 is virtual time at t_1 and v_2 is virtual time at t_2.

Lemma 1 is true since at t_2 any backlogged flow has dispatched all requests whose start tag $\leq v_2 - T$. Only the last request may be outstanding at t_2—i.e., all but the last request must have completed. Since the last request's size is at most l_f^{max}, the finish tag of the last completed request must be at least $v_2 - T - l_f^{max}/r_f$. Therefore if we just count the completed requests in $[t_1, t_2]$, the minimum service received by backlogged flow f is at least $r_f \cdot (v_2 - T - v_1) - l_f^{max}$.

Lemma 2 (Upper bound of received service by a flow): *If flow f is backlogged throughout the internal $[t_1, t_2]$, then in an MQFQ(1) system with throttling threshold T:*

$$W_f(t_1, t_2) \leq r_f \cdot (v_2 + T - v_1) + l_f^{max}$$

Lemma 2 is true since at t_2 flow f may have, at most, dispatched all requests with start tag $\leq v_2 + T$. In the maximum case, the last completed request's finish tag will be no more than $v_2 + T$. In addition, one more request of size at most l_f^{max} may be outstanding and, in the maximum case, almost entirely serviced. Counting the completed requests and the outstanding request, the maximum service received by flow f is at most $r_f \cdot (v_2 + T - v_1) + l_f^{max}$.

Unfairness is maximized when one flow receives its upper bound of service while another flow receives its lower bound. Therefore, unfairness in MQFQ(1) with throttling threshold T is bounded by

$$\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right| \leq 2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

This completes the proof of Theorem 1. ■

Note that when $T = 0$, MQFQ(1) provides the same fairness bound as SFQ. Therefore T represents a tradeoff between fairness and scalability in a multi-queue system.

If we allow $D > 1$ parallel dispatches in an MQFQ(D) server, the fairness bound changes as follows:

Theorem 2 *In any interval for which flows f and m are backlogged during the entire interval, the difference of weighted services received by the two flows at an MQFQ(D) server with throttling threshold T is given as:*

$$\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right| \leq (D+1) \left(2T + \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m} \right)$$

This is true based on a combination of Theorem 1 and the proved fairness bound for SFQ(D) [23]. We omit the detailed proof. When the throttling threshold $T = 0$, MQFQ(D) provides the same fairness bound as SFQ(D).

3 Scalability

MQFQ employs a separate priority queue for every CPU (hardware thread), to minimize coherence misses and maximize scalability. A certain amount of sharing and synchronization is required, however, to maintain fairness across queues. Specifically, we need to track (1) the progression of virtual time; (2) the number of available I/O slots and the queues that can use them; and (3) the state of queues (throttled or not) and when they should be unthrottled. Our guiding principle is to maximize locality wherever possible. So long as utilization and fairness goals are not violated, we prefer to dispatch from the local queue, queues on the same core, queues on the same socket, and queues on another socket, in that order.

3.1 Virtual Time

Virtual time in MQFQ reflects resource usage (e.g., bandwidth consumed), and not wall-clock time. When a flow transitions from lagging to backlogged, the request responsible for the transition is set to have its start tag equal to current virtual time. As long as the flow remains backlogged, its following requests get increasing start tags with respect to the flow's resource usage: the start tag of each new request is set to the end tag of the previous request. Virtual time, in turn, is the minimum start tag of any request across all queues.

Naively, one might imagine an array, indexed by queue, with each slot indicating the start tag of its queue's lead request (if any). We could then compute the global virtual time by scanning the array. Such a scan, however, is far too expensive to perform on a regular basis (see Sec. 4.3.1). Instead, we use an instance of Liu et al.'s *mindicator* structure [29], modified to preclude decreases in the global minimum. The mindicator is a tree-based structure reminiscent of a priority-queue heap. Each queue is assigned a leaf in the tree; each internal node indicates the minimum of its children's values. A flow whose virtual time changes updates its leaf and, if its previous value was the minimum among its siblings, propagates the update root-ward. Changes reach the root only when the global minimum changes. While this is not uncommon (time continues to advance, after all), many requests in a highly parallel device complete a little out of order, and the mindicator achieves a significant reduction in contention.

Within each flow, we must also track the largest finish tag across all threads. For this we currently employ a simple shared integer value, updated (with fetch-and-add) on each request dispatch. In future work, we plan to explore whether better performance might be obtained with a scalable monotonic counter [6, 14], at least for flows with many threads.

3.2 Available Slots

A queue in MQFQ is unable to dispatch either when it is too far ahead of virtual time or when the device is saturated. For the latter case, MQFQ must track the number of outstanding (dispatched but not yet completed) requests on the device. Ideally, we want to dispatch exactly as many requests as the

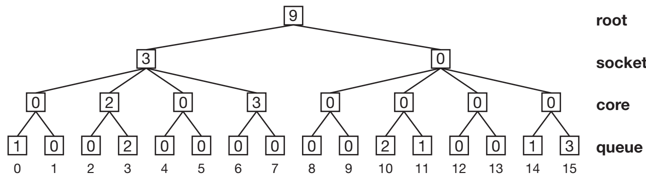


Figure 3: Example token tree for a 2-socket, 4-core-per, 2-thread-per machine. Values indicate currently unused device capacity. (If the device were fully subscribed [D outstanding requests], all values in the tree would be zero.) In the figure, there are 3 slots immediately available to queue 15. Queues 6 or 7 could use capacity allocated to their core; queues 4 or 5 could use capacity allocated to their socket; queues 8 or 9 would need to use capacity from the root.

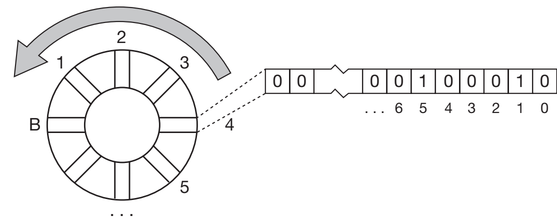


Figure 4: Timer wheel for throttled queues. If queue q is $k > T$ units ahead of global virtual time, it is placed in bucket $\min(\lceil k/b \rceil, B)$, where B is the number of buckets and b is a quantization parameter. In the figure, queues 1 and 5 are throttled in bucket 4.

device can handle in parallel, thereby avoiding any buildup in the device and preserving our ability to choose the ideal request to submit when an outstanding request completes.

We find (see Sec. 4.3.3) that a naive single shared cache line, atomically incremented and decremented upon dispatch and completion of requests, fails to scale when many queues are frequently trying to update its value. We therefore aim to improve locality by preferentially allocating available slots to physically nearby queues, in a manner reminiscent of cohort locks [15]. This approach meshes well with our notification mechanism, which prefers to unblock nearby queues.

As a compromise between locality and flexibility, we have implemented a structure we call the *token tree* (Fig. 3). Values in leaves represent unused capacity ("slots") currently allocated to a given local queue. Parent nodes represent additional capacity allocated to a given core, and so on up the tree. The values of all nodes together sum to the difference between D and the number of active requests on the device. When we need to dispatch a request, we try to acquire a slot from the leaf associated with the local queue. If the leaf is zero, we try to fetch from its parent, continuing upward until we reach the root. If nothing is available at that level, we suspend the queue. If there is unused capacity elsewhere in the tree, queues in that part of the tree will eventually be throttled. Capacity will then percolate upward, and ready queues will be awoken.

When releasing slots (in the completion interrupt handler, when the local queue is throttled or empty), we first choose a queue to awaken. We then release slots to the lowest common ancestor (LCA) of the local and the target CPUs in the token tree. Finally, we awaken the target CPU with an interprocessor interrupt (IPI). The strategy of picking nearby queues tends to keep capacity near the leaves of the token tree, minimizing contention at the higher levels, minimizing the cost of the IPI, and maximizing the likelihood that slots will be passed through a cache line in a higher level of the memory hierarchy. Experiments described in Sec. 4.3.2 confirm that IPIs significantly outperform an alternative based on polling.

3.3 Ready and Throttled Queues

The D parameter in MQFQ controls the number of outstanding requests and is a trade-off between utilization and fairness. While a larger D may better utilize the device, it can also impose looser fairness bounds and higher waiting time for incoming requests from a slower flow. Therefore, MQFQ will stop dispatching once there are D outstanding requests in the device. A queue in this case is likely to be both nonempty and unthrottled; such a queue is said to be *ready*.

As noted in Sec. 3.2, a completion handler whose local queue is empty or throttled will give away its released token. To do so, it looks for the closest queue (based on a pre-computed proximity matrix) that is flagged as ready and passes the token through the token tree.

Regardless of the number of outstanding requests, a queue will be throttled when its lead request is T ahead of global virtual time. When this happens, we need to be able to tell when virtual time has advanced enough that the queue can be unthrottled. To support this operation, we employ a simple variant of the classical timer wheel structure [43, 47] (Fig. 4). Each *bucket* of the wheel represents a b-unit interval of virtual time, and contains (as a bitmask) the set of queues that should be unthrottled at the beginning of that interval. Conceptually, we turn the wheel every b time units (in actuality, of course, we update an index that identifies bucket number 1), clear the bitmask in the old bucket 1, and unthrottle the queues that used to appear in that mask.

Given a finite number of buckets, B, a queue that needs to be throttled for longer than $B \times b$ will be placed in bucket B; this means that the wakeup handler for a queue must always double-check to make sure it doesn't have to throttle the queue again. Unlike a classical timing wheel, which contains a list of timer events in every bucket, our bitmask buckets can be manipulated with compare-and-swap, making the whole wheel trivially nonblocking.

As noted in Sec. 3.2, when slots become available in a completion handler, we choose queues from among the ready set, release the slots to the token tree, and send IPIs to the CPUs of the chosen queues. In a similar vein, if slots are available at the root of the token tree when the throttling wheel is turned, we likewise identify ready queues to which to send

IPIs. No fairness pathology arises in always choosing nearby queues: if some far-away queue lags too far behind, nearby queues will end up throttling, slots will percolate upward in the token tree, and the lagging queues will make progress.

3.4 Determining D and T in Practice

In practice, we use a hand-curated workload with varying degrees of concurrency and request sizes (with an approach similar to that of Chen et al. [9]) as a one-time setup step to discover the internal parallelism of a given multi-queue device which determines the parameter D. Any smaller value for D will not saturate the device, while larger Ds would lead to greater unfairness – specially for burstier workloads.

Unlike D which is determined solely by the degree of parallelism in the multi-queue device, the parameter T is affected by the characteristic of the workload – i.e., concurrency and request size. While a single-threaded workload can afford to have $T = 0$, a workload with small requests being submitted from multiple threads across multiple sockets require larger T value. To that end, once we have determined D, in a one-time setup step, we *over-provision* the parameter T for the worst-case workload so that the maximum throughput of the device can always be met.

4 Evaluation

We evaluate fairness and performance of MQFQ on two fast, multi-queue devices: NVMe over RDMA (NVMf, with multi-queue NICs) and multi-queue SSD (MQ-SSD). We also evaluate the scalability of each of our concurrent data structures.

In our NVMf setup, the host machine (where MQFQ runs) issues NVMe requests over RDMA to the target machine, which serves the requests directly from DRAM. We use the kernel host stack and SPDK [21] target stack. This setup can reach nearly 4 M IOP/s for 1KB requests. In our MQ-SSD setup, requests are fulfilled by a PCIe-attached Intel P3700 NVMe MQ-SSD. This setup provides nearly 0.5 M IOP/s for 4K requests.

We measured (with an approach similar to that of Chen et al. [9]) available internal parallelism to be 128 for the NVMf setup and 64 for the MQ-SSD setup. We chose T in each setup to be (roughly) the smallest value that didn't induce significant contention. We preconditioned the MQ-SSD with sequential writes to the whole address space followed by a series of random writes to reach steady state performance. We also disabled power management to ensure consistent results. We ran all experiments on a Linux 4.15 kernel in which KPTI [12] was disabled via boot parameter. For scalability experiments, thread affinities were configured to fill one hardware thread on each core of the first socket, then one on each core of the second socket before returning to populate the second hardware thread of each core. The CPU mask for fairness experiments was configured to partition the cores among competing tasks. Table 1 summarizes the experimental setup. In all of the experiments we use the length of requests in KB to advance virtual time—hence the unit for T is KB. Because the MQ-SSD setup has significantly lower bandwidth than the NVMf setup, we use it only for fairness experiments, not for scalability. The source code for our implementation is available at `http://github.com/hedayati/mqfq`.

4.1 Fairness and Efficiency

We compare MQFQ to two existing systems: (1) the recommended Linux setup for fast parallel devices, which performs no I/O scheduling (`nosched`) and is thus contention free, and (2) Budget Fair Queueing (BFQ) [46], a proportional share scheduler included for multi-queue stacks since Linux 4.12. For each of these, we consider three benchmarks: (a) the Flexible I/O Tester (FIO) [3], (b) the FlashX graph processing framework [52], and (c) the Aerospike key-value store [41].

FIO: FIO is a microbenchmark that allows flexible configuration of I/O patterns and scales quite well. We use FIO to generate workloads with known characteristics. Because FIO does so little processing per request, we also use it as an antagonist in multiprogrammed runs with FlashX and Aerospike. Each FIO workload has a name of the form αxβ (e.g., `2x4K`) where α indicates the number of threads (each on a dedicated queue) and β indicates the size of each request. For proportional sharing tests, we also indicate the weight of the flow in parentheses (e.g., `2x4K(3)`). The FIO queue depth (i.e., the number of submitted but not yet completed requests) is set to 128—large enough to maintain maximum throughput to the device.

To evaluate fairness and efficiency, we consider co-runs of FIO workloads where the internal device scheduler (if any) fails to provide fairness. We compare the slowdown of the flows relative to their time when running alone (in the absence of resource competition) as a measure of fairness as well as aggregated throughput as a measure of efficiency. We explore three cases in which competing flows differ in only one characteristic—request size, concurrency, or priority (weight). The results show that the underlying request processing, being oblivious to these characteristics, fails to provide fairness.

In Fig. 5 top-left and bottom-left, each of the flows uses an equal number of device queues. The device alternates between queues and guarantees the same number of processed requests from each. This results in flows sharing the device in proportion to request sizes rather than getting equal shares.

Fig. 5 top-middle and bottom-middle show two flows, one of which uses half the number of physical queues used by the other flow. With both flows submitting 4KB requests, the requests are processed in proportion to the number of utilized queues, causing unfairness.

Finally, Fig. 5 top-right and bottom-right show how MQFQ can be used to enforce shares in proportion to externally-specified per-flow weights (shown in parentheses).

In all of the above cases, the BFQ scheduler also guarantees fairness (as defined by flows' throughputs) but at a much higher cost compared to MQFQ.

Table 1: Experimental setup.

	MQ-SSD Setup	NVMf Setup
CPU & Mem.	Intel E5-2620 v3 (Haswell) @ 2.40GHz – 8GB	Intel E5-2630 v3 (Haswell) @ 2.40GHz – 64GB
Sockets×Cores	2×6 (24 hardware threads)	2×8 (32 hardware threads)
Target device	Intel P3700 NVMe MQ-SSD (800GB)	NVMe over RDMA Mellanox ConnectX-3 VPI Dual-Port 40Gbps
MQFQ parameters	$D = 64$, $T = 45$KB	$D = 128$, $T = 64$KB

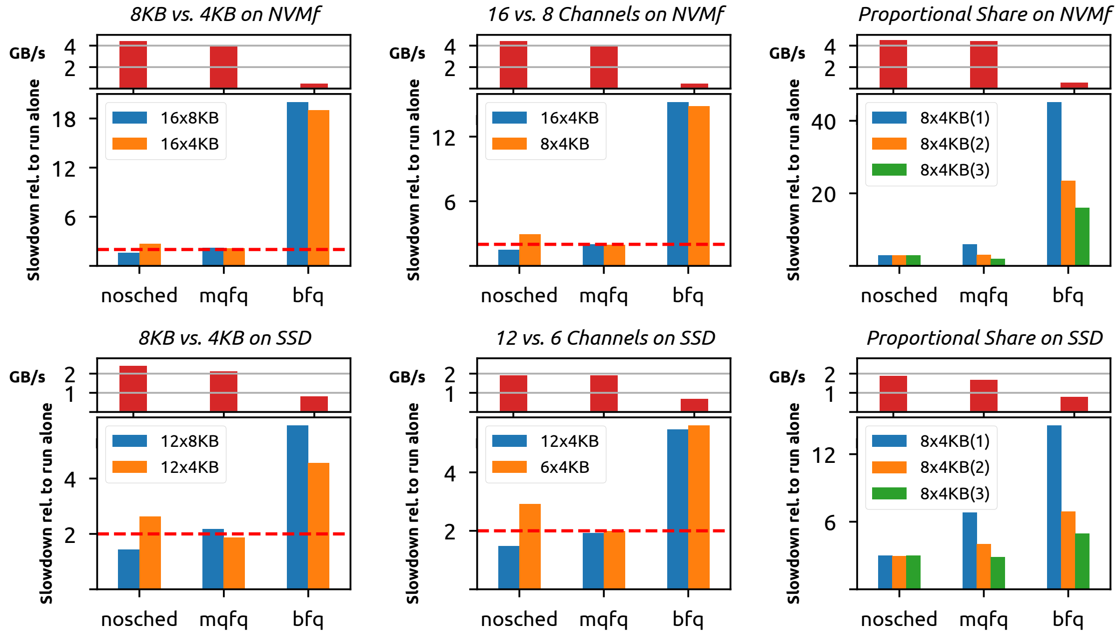


Figure 5: FIO fairness and efficiency. Round-robin (`nosched`) processing is unfair with respect to different request sizes (left), different numbers of queues (middle) and different proportional shares (right). Red dashed lines in the left and middle columns indicate proportional (ideal) slowdown. Aggregate bandwidth is shown above each graph.

FlashX: FlashX is a data analytics framework that utilizes SSDs to scale to large datasets. It can efficiently store and retrieve large graphs and matrices, and uses FlashR, an extended R programming framework, to process terabyte-scale datasets in parallel. We used FlashX to execute pagerank on the SOC-LiveJournal1 social network graph from SNAP [28]. The graph has 4.8M vertices and 68.9M edges and is stored on SSD or the NVMf target's DRAM for corresponding tests. We use FIO as an antagonist process to create contention with FlashX over the storage resource.

Fig. 6 shows the slowdown of co-runs of FlashX and FIO with different schedulers (or none—`nosched`). FlashX does not maintain a large queue depth; as a result, it can sustain only a fraction of the device's throughput. FIO, by contrast, is able to fully utilize the device given its large (I/O) parallelism. Running these together, MQFQ guarantees that FlashX gets its small share of I/O, while the rest is available to FIO, resulting in small (better than proportional) slowdowns (33% for FlashX and 14% for FIO on average between MQ-SSD and NVMf) — note that this is not unexpected since one of the flows, i.e., FlashX, is not saturating. While BFQ also reduces the slowdown for FlashX (from almost 4× to less than 2×), it slows down FIO due to its lack of support for I/O parallelism.

Aerospike: Aerospike is a flash-optimized key-value store. It uses direct I/O to a raw device in order to achieve high performance. Meta-data is kept in memory, but we configure our instance to make sure all requests will result in an I/O to the underlying device. We use the benchmark tool provided with Aerospike, running on a client machine, to drive a workload of small (512B) reads, ensuring that there will be no contention over the network for the NVMf setup. As in the FlashX experiments, we use FIO as a competitor workload.

Fig. 7 shows the slowdown of co-runs of Aerospike and FIO under BFQ, MQFQ, and `nosched`. For the NVMf setup, despite performing nearly 1 M transactions /sec., Aerospike fails to saturate the device before running out of CPUs. Therefore, as with FlashX, the co-run under MQFQ has negligible slowdown (3% for Aerospike and less than 20% for FIO). However, on the MQ-SSD setup Aerospike can fully utilize the

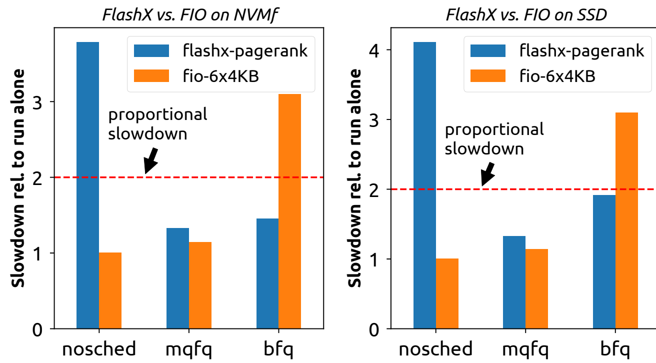


Figure 6: Fairness comparison for FlashX. MQFQ maintains fairness for FlashX, while allowing FIO to utilize the remaining bandwidth of the device.

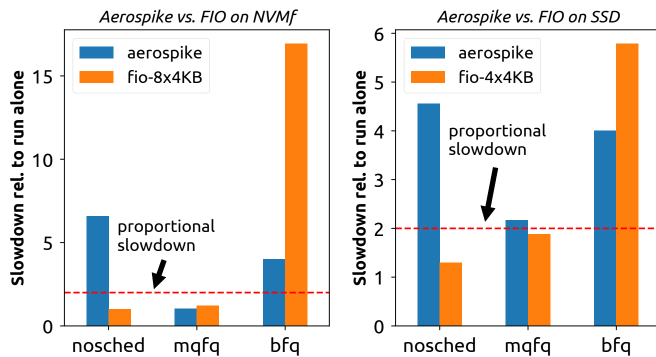


Figure 7: Fairness comparison for Aerospike. MQFQ maintains fairness (approximately at or below proportional slowdown). On the MQ-SSD (right), where Aerospike can utilize the full device, FIO is slowed down to half the available bandwidth.

device (with nearly 0.5 M transactions /sec.) and Aerospike and FIO end up getting half the available bandwidth each. BFQ's lack of support for parallel dispatch is evident on the faster NVMf device, where it results in 15× slowdown for FIO while giving only a modest improvement for Aerospike.

4.2 Scalability

We compare the scalability of MQFQ to that of an existing single-queue implementation of fair queueing—i.e., BFQ [46]. As noted in Sec. 1, Linux BFQ doesn't support concurrent dispatches and may not be able to fully utilize a device with internal parallelism. Other schedulers with support for parallel dispatch (e.g., FlashFQ [40]) have no multi-queue implementation. As a reasonable approximation of the missing strategy, we also compare MQFQ to a modified version of itself (MQFQ-serial) that serializes dispatches using a global lock. It differs from a real single-queue scheduler for a device with internal parallelism in that it maintains the requests in separate, per-CPU queues coordinated with our scalable data structures and the T and D parameters.

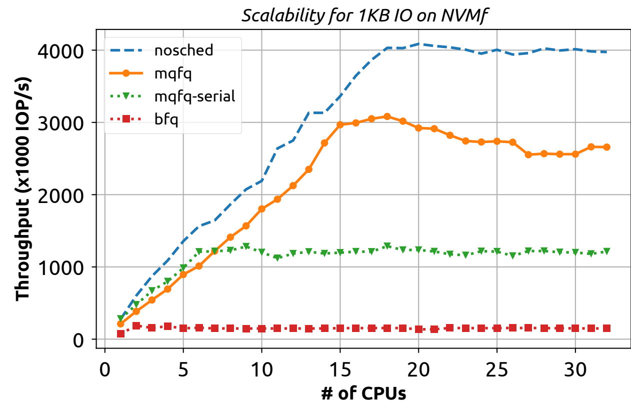


Figure 8: Overall scalability of unfair Linux multi-queue vs. MQFQ vs. MQFQ-serial vs. BFQ.

Our MQ-SSD setup at 460K IOP/s is not suitable for scalability experiments—the IOP/s limit, rather than the scheduler, becomes the scaling bottleneck. Some higher-IOP/s devices exist in the market and more will surely emerge in the future. Employing an array of SSDs can also enable over a million IOP/s. Alternatively, remote storage software solutions (e.g., ReFlex [26], NVMe over Fabric [33], FlashNet [45]) have the potential to yield more than a million IOP/s.

For this scalability evaluation, we therefore rely on the NVMf setup with 1KB requests. We chose 1KB because it yields the largest number of IOP/s (more request churn, leading to higher scheduler contention). In the `nosched` (no contention) case, this setup can reach 4 M IOP/s. We need multiple FIO threads to reach this maximum throughput.

Fig. 8 compares the throughput achieved with `nosched`, MQFQ, MQFQ-serial, and BFQ. With 15–19 active threads, MQFQ reaches more than 3 M IOP/s—2.6× better than MQFQ-serial and 20× better than BFQ. This constitutes 71% of the peak throughput of the in-memory NVMf device while providing the fairness properties needed for shared systems (as demonstrated in Sec. 4.1).

4.3 Design Decisions and Parameters

We assess the degree to which each of MQFQ's scalable data structures improves performance.

4.3.1 Virtual Time

We first evaluate the scalability of computing virtual time in MQFQ. As described in Sec. 3.1, our implementation uses a variant of the mindicator [29] to find the smallest start tag among queued requests across all queues. As in the token tree (Fig. 3), we structure the mindicator with successive levels for cores, sockets, and the full machine.

Fig. 9 shows how the mindicator scales with the number of queues. We are unaware of any existing data structure suitable as a replacement for the mindicator; we therefore implemented another lock-free alternative in which the minimum is

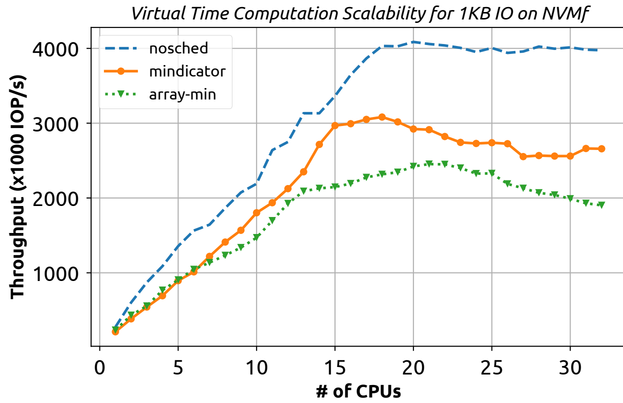


Figure 9: Throughput when maintaining virtual time with a mindicator vs. iterating over an array of queue minima.

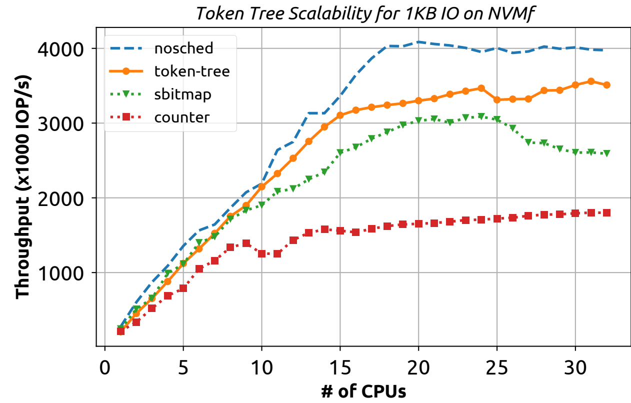


Figure 11: Scalability of token-tree vs. global counter vs. scalable bitmap in maintaining available dispatch slots.

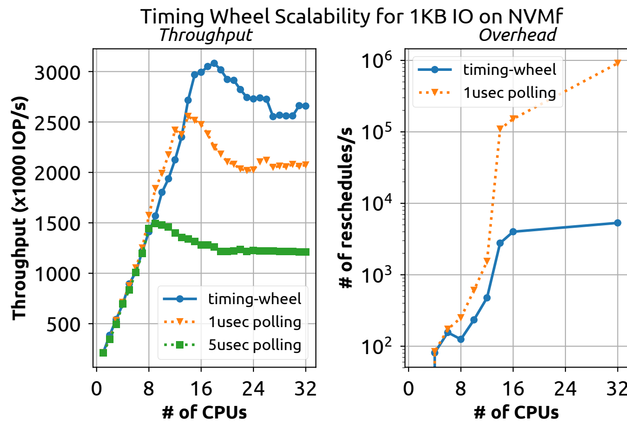


Figure 10: Scalability of unthrottling. Left: MQFQ throughput achieved using timing wheel vs. $1\mu s$ and $5\mu s$ polling. Right: polling causes spurious reschedules.

found by iterating over an array of queue-local minima after each request dispatch. (This could be thought as a one-level mindicator.) Our contention-localizing structure outperforms the array scan by nearly 40% at 32 threads.

4.3.2 Unthrottling

As discussed in Sec. 2, when a queue cannot dispatch it will be throttled. Once the situation changes (completion or progress of virtual-time) some throttled queues may need to be unthrottled. Any delay in doing so could leave the device underutilized. Our approach uses inter-processor interrupts to promptly notify appropriate CPUs that they can proceed when the unthrottling condition is met. We use a scalable timer wheel (Sec. 3.3) to support such notifications efficiently.

For comparison, arranging for each queue to poll the condition would be an easy but expensive way to implement unthrottling. We explore this option with a pinned, high resolution timer (hrtimer [11]), as it requires no communication between queues and can provide latency comparable to that of a cross-socket inter-processor interrupt. The timer is armed whenever the queue is throttled and upon firing, reschedules the dispatch routine. The effect is essentially polling for a change in virtual time, with a polling frequency determined by the value of the timer.

Fig. 10 (left) compares the throughput that MQFQ can achieve with the timing wheel vs. polling at $1\mu s$ or $5\mu s$ intervals. Results confirm that a delay in unthrottling leads to throughput degradation. Even extremely frequent ($1\mu s$) polling cannot achieve IOP/s performance comparable to that of our timer wheel approach. Less frequent polling leads to a dispatch delay that leaves the device underutilized.

In order to quantify the wasted CPU, we measure the number of reschedule operations caused by our timer wheel and by $1\mu s$ polling. The difference between the two shows how inefficient polling can be (the timer wheel incurs no spurious reschedules). Fig. 10 (right) shows the savings, in reschedules per second, achieved by using the timer wheel instead of a $1\mu s$ timer. With a few CPUs, roughly every queue is being signaled on every completion (so a carefully chosen frequency for polling that matches the rate of completion could be practical when the device is fully utilized), but the number of wasted cycles grows with the number of CPUs. With the timing wheel, on the other hand, unthrottling comes only as a result of completion, and therefore is upper-bounded by the device throughput.

4.3.3 Dispatch Slots

In order to keep a device with internal parallelism fully utilized, while also avoiding queue build-up in the device (which would adversely affect the fairness guarantee), MQFQ has to track the number of available dispatch slots. This number is modified by each queue as a result of a dispatch or a completion. Our scalable MQFQ design uses a novel token tree data structure for this purpose (presented in Sec. 3.2).

Kyber [39], a multi-queue I/O scheduler added since Linux 4.12, uses another data structure, called *sbitmap* (for Scalable

Bitmap), to throttle asynchronous I/O operations if the latency experienced by synchronous operations exceeds a threshold. The main idea in sbitmap is to distribute the tokens as bits in a number of cache lines (determined by expected contention over acquiring tokens). A thread tries to find the first clear bit in the cache line where the last successful acquire happened, falling back to iterating over all cache lines if all bits are set. This data structure reduces contention when the number of tokens is significantly larger than the number of threads. Yet another alternative to maintain a single global count of available dispatch slots using atomic increments and decrements.

Fig. 11 plots 1KB MQFQ IOP/s as a function of thread count using an atomic counter, a scalable bitmap, and a token tree to track the number of dispatched requests. To isolate the impact of these data structures, we disable virtual time computation in MQFQ. Using an atomic counter doesn't scale beyond the first socket. The scalable bitmap falls short when the number of waiting requests is significantly larger than device parallelism, resulting in local acquire and release of tokens. In comparison, the token tree paired with our throttling mechanism prefers interaction with local queues (based on a pre-computed proximity matrix) as long as they are no more than T ahead of virtual time, resulting in significantly better scalability (more than $2\times$ the throughput of the atomic counter and 36% more than the scalable bitmap).

5 Related Work

Fairness-oriented resource scheduling has been extensively studied in the past. Lottery scheduling [49] achieves probabilistic proportional-share resource allocation. Fairness can also be realized through per-task timeslices as in Linux CFQ [2] and BFQ [46], Argon [48], and FIOS [35]. Timeslice schedulers, however, are generally not *work-conserving*: they will sometimes leave the device unused when there are requests available in the system. The original fair queueing approaches, including Weighted Fair Queueing (WFQ) [13], Packet-by-Packet Generalized Processor Sharing (PGPS) [34], and Start-time Fair Queueing (SFQ) [18], employ virtual-time–controlled request ordering across per-flow request queues to maintain fairness.

Fair queueing approaches like SFQ(D) [23] and FlashFQ [40] have been tailored to manage I/O resources, allowing requests to be re-ordered and dispatched concurrently for better I/O efficiency in devices with internal parallelism. To maintain fairness in a multi-resource (e.g., CPU, memory and NIC) environment, DRFQ [16] adapted fair queueing by tracking usage of the respective dominant resource of each operation. Disengaged fair queueing [30] emulates the effect of fair queueing on GPUs while requiring only infrequent OS kernel involvement. It accomplishes its goal by monitoring and mitigating potential unfairness through occasional traps. All previous fair queueing schedulers assume a serializing scheduler over a single device queue, which does not scale well on modern multicores with fast multi-queue devices.

For multi-queue SSDs, Ahn et al. [1] supported I/O resource sharing by implementing a bandwidth throttler at the Linux cgroup layer (above the multi-queue device I/O paths). However, their time interval budget-based resource control is not work conserving: if one cgroup does not use its allotted resources in an interval, those resources are simply wasted. Lee et al. [27] improved read performance by isolating queues of multi-queue SSDs used for reads from those used for writes. Kyber [39] achieves better synchronous I/O latency by throttling asynchronous requests. However, neither approach is a full solution for fair I/O resource management. Stephens et al. [42] found that the internal round-robin scheduling of hardware queues in NICs leads to unfairness when the load is asymmetrically distributed across a NIC's multiple hardware queues. Their solution, Titan, requires programmable NICs to internally implement deficit round-robin and service queues in proportion to configured weights. FLIN [44] identifies major sources of interference in multi-queue SSDs and implements a scheduler in SSD controller firmware to protect against them. Unlike MQFQ, which is applicable to accelerators and multi-queue NICs, FLIN deals with the idiosyncrasies of Flash devices such as garbage collection and access patterns. In addition, FLIN considers any request originating from the same host-side I/O queue as belonging to the same "flow" and, being implemented in hardware, is unable to reason in terms of system-level resource principals (applications, virtual machines, or Linux cgroups).

For performance isolation and quality-of-service, ReFlex [26] employs a per-tenant token bucket mechanism to achieve latency objectives in a shared-storage environment. The token bucket mechanism and fair queueing resource allocation are complementary—the former performs admission control under a given resource allocation while the latter supports fair, work-conserving resource uses. Decibel [31] presents a system framework for resource isolation in rack-scale storage but it does not directly address the problem of resource scheduling. It uses two existing scheduling policies in its implementation—strict time sharing is not work-conserving; deficit round robin is work-conserving but requires a serializing scheduler queue that limits scalability.

Among multicore operating systems, Arrakis [36] and IX [4] support high-speed I/O by separating the control plane (managed by the OS) and the data plane (bypassing the OS) to achieve coherency-free execution. Their OS control planes enforce access control but not resource isolation or fair resource allocation. Zygos [37] suggests that sweeping simplification introduced by shared-nothing architectures like IX [4] leads to (1) not being work-conserving and (2) suffering from head-of-the-line blocking. They propose a work-stealing packet processing scheme that, while introducing cross-core interactions, eliminates head-of-the-line blocking and improves latency. Recent work has also built scalable data structures that localize synchronization in the multicore memory hierarchy (intra-core rather than inter-core; intra-socket rather

than inter-socket). Examples include the mindicator global minimum data structure [29], atomic broadcast trees [24], and NUMA-aware locks [15] and data structures [7]. For MQFQ, we introduce new scalable structures, including a timer wheel to track virtual time indexes and a token tree to track available device dispatch slots.

6 Conclusion

With the advent of fast devices that can complete a request every microsecond or less, it has become increasingly difficult for the operating system to fulfill its responsibility for fair resource allocation—enough so that some OS implementations have given up on fairness altogether for such devices. Our work demonstrates that surrender is not necessary: with judicious use of scalable data structures and a reformulation of the analytical bounds, we can maintain fairness in the long term and bound it in the short term, all without compromising throughput.

Our formalization of multi-queue fair queueing introduces a parameter, T, that bounds the amount of service that a flow can receive in excess of its share. Crucially, this bound does not grow with time. Moreover, our new definition of virtual time is provably equivalent to existing definitions when T is set to zero. Experiments with a modified Linux 4.15 kernel, a two-socket server, and a fast NVMe over RDMA device confirm that MQFQ can provide both fairness and very high throughput. Compared to running without a fairness algorithm on an NVMf device, our MQFQ algorithm reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload. Its peak throughput reaches 3.1 Million IOP/s on a single machine, outperforming a serialized version of our own algorithm by 2.6× and Linux BFQ by 20×.

In future work, we plan to develop strategies for automatic tuning of the T and D parameters; extend our implementation to handle small computational kernels for GPUs and accelerators; and evaluate the extent to which fairness guarantees can continue to apply even to kernel-bypass systems, with dispatch queues in user space.

Acknowledgment

We thank our shepherd, Jian Huang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CNS-1319417, CCF-1717712, CCF-1422649 and by a Google Faculty Research award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] S. Ahn, K. La, and J. Kim. Improving I/O resource sharing of Linux cgroup for NVMe SSDs on multi-core systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Denver, CO, June 2016.

[2] J. Axboe. Linux block IO—Present and future. In *Ottawa Linux Symp.*, pages 51–61, Ottawa, ON, Canada, July 2004.

[3] J. Axboe et al. Flexible I/O tester. github.com/axboe/fio.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, Oct. 2014.

[5] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *6th ACM Intl. Systems and Storage Conf. (SYSTOR)*, Haifa, Israel, June 2013.

[6] 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 *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, BC, Canada, 2010.

[7] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, Xi'an, China, Apr. 2017.

[8] B. Caldwell. Improving block-level efficiency with *scsi-mq*. *arXiv e-prints*, abs/1504.07481v1, Apr. 2015.

[9] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 266–277, San Antonio, TX, 2011.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 1–17, Farminton, PA, 2013.

[11] J. Corbet. The high-resolution timer API. lwn.net/Articles/167897.

[12] J. Corbet. The current state of kernel page-table isolation. lwn.net/Articles/741878/, Dec. 2017.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12, Austin, TX, Sept. 1989.

[14] D. Dice, Y. Lev, and M. Moir. Scalable Statistics Counters. In *25th ACM Symp. on Parallelism in*

Algorithms and Architectures (SPAA), pages 43–52, Montreal, PQ, Canada, 2013.

[15] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. on Parallel Compututing*, 1(2):13:1–13:42, Feb. 2015.

[16] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 1–12, Helsinki, Finland, 2012.

[17] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *13th IEEE Conf. on Networking for Global Communications (INFOCOM)*, pages 636–646, San Jose, CA, 1994.

[18] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[19] A. G. Greenberg and N. Madras. How fair is fair queueing. *Journal of the ACM*, 39(3):568–598, July 1992.

[20] Hyper-Q Example. developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[21] Intel Corp. Storage performance development kit. www.spdk.io.

[22] Intel Optane SSD DC P4800X Series. www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-aic.html.

[23] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for a Storage Service Utility. In *Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 37–48, New York, NY, 2004.

[24] S. Kaestle, R. Achermann, R. Haecki, M. Hoffmann, S. Ramos, and T. Roscoe. Machine-aware atomic broadcast trees for multicores. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 33–48, Savannah, GA, Nov. 2016.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conf. (ATC)*, pages 437–450, Denver, CO, June 2016.

[26] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash $\approx$ local flash. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 345–359, Xi'an, China, Apr. 2017.

[27] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom. Improving read performance by isolating multiple queues in NVMe SSDs. In *11th Intl. Conf. on Ubiquitous Information Management and Communication*, Beppu, Japan, Jan. 2017.

[28] J. Leskovec and A. Krevl. SNAP datasets: Stanford large network dataset collection. snap.stanford.edu/data/.

[29] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A Scalable Approach to Quiescence. In *2013 IEEE 33rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 206–215, Philadelphia, PA, July 2013.

[30] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, Mar. 2014.

[31] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 17–33, Boston, MA, Mar. 2017.

[32] Nvidia Corp. Sharing a GPU between MPI processes: Multi-process service (MPS). docs.nvidia.com/deploy/mps/index.html.

[33] NVM Express Workgroup. NVM express, revision 1.3a. nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf, Oct. 2017.

[34] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1992.

[35] S. Park and K. Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *10th USENIX Conf. on File and Storage Technologies (FAST)*, pages 13–13, San Jose, CA, 2012.

[36] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the control plane. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[37] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *26th Symp. on Operating Systems Principles (SOSP)*, pages 325–341, Shanghai, China, 2017.

[38] Samsung SSD PM1725a. www.samsung.com/semiconductor/global/file/insight/2016/08/Samsung_PM1725a-1.pdf.

[39] O. Sandoval. Kyber multi-queue I/O scheduler. lwn.net/Articles/720071/.

[40] K. Shen and S. Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX Annual Technical Conf. (ATC)*, San Jose, CA, June 2013.

[41] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic. Aerospike: Architecture of a real-time

operational dbms. *Proc. of the VLDB Endowment*, 9(13):1389–1400, Sept. 2016.

[42] B. Stephens, A. Singhvi, A. Akella, and M. Swift. Titan: Fair packet scheduling for commodity multiqueue NICs. In *USENIX Annual Technical Conf. (ATC)*, pages 431–444, Santa Clara, CA, 2017.

[43] S. A. Szygenda, C. W. Hemming, and J. M. Hemphill. Time flow mechanisms for use in digital logic simulation. In *5th ACM Winter Simulation Conf.*, pages 488–495, New York, NY, 1971.

[44] A. Tavakkol, M. Sadrosadati, S. Ghose, J. S. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives. In *45th Intl. Symp. on Computer Architecture (ISCA)*, pages 397–410, Los Angeles, CA, 2018.

[45] A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, I. Koltsidas, K. Kourtis, and T. R. Gross. Flashnet: Flash/network stack co-design. In *10th ACM Intl. Systems and Storage Conf. (SYSTOR)*, pages 15:1–15:14, Haifa, Israel, 2017.

[46] P. Valente and A. Avanzini. Evolution of the BFQ Storage-I/O scheduler. algo.ing.unimo.it/people/paolo/disk_sched/mst-2015.pdf.

[47] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *ACM/IEEE Trans. on Networking*, 5(6):824–834, Dec. 1997.

[48] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conf. on File and Storage Technologies (FAST)*, pages 61–76, San Jose, CA, Feb. 2007.

[49] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *1st USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, Nov. 1994.

[50] Skyhawk & Skyhawk Ultra NVMe PCIe SSD. www.sandisk.com/content/dam/sandisk-main/en_us/assets/resources/data-sheets/Skyhawk-Series-NVMe-PCIe-SSD-DS.pdf.

[51] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *8th ACM Intl. Systems and Storage Conf. (SYSTOR)*, Haifa, Israel, May 2015.

[52] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conf. on File and Storage Technologies (FAST)*, pages 45–58, Santa Clara, CA, 2015.

BRAVO – Biased Locking for Reader-Writer Locks

Dave Dice
Oracle Labs
dave.dice@oracle.com

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Abstract

Designers of modern reader-writer locks confront a difficult trade-off related to reader scalability. Locks that have a compact memory representation for active readers will typically suffer under high intensity read-dominated workloads when the "reader indicator" state is updated frequently by a diverse set of threads, causing cache invalidation and coherence traffic. Other designs use distributed reader indicators, one per NUMA node, per core or even per thread. This improves reader-reader scalability, but also increases the size of each lock instance and creates overhead for writers.

We propose a simple transformation, BRAVO, that augments any existing reader-writer lock, adding just two integer fields to the lock instance. Readers make their presence known to writers by hashing their thread's identity with the lock address, forming an index into a *visible readers table* and installing the lock address into the table. All locks and threads in an address space can share the same readers table. Crucially, readers of the same lock tend to write to different locations in the table, reducing coherence traffic. Therefore, BRAVO can augment a simple compact lock to provide scalable concurrent reading, but with only modest and constant increase in memory footprint.

We implemented BRAVO in user-space, as well as integrated it with the Linux kernel reader-writer semaphore (`rwsem`). Our evaluation with numerous benchmarks and real applications, both in user and kernel-space, demonstrate that BRAVO improves performance and scalability of underlying locks in read-heavy workloads while introducing virtually no overhead, including in workloads in which writes are frequent.

1 Introduction

A reader-writer lock, also known as a shared-exclusive lock, is a synchronization primitive for controlling access by multiple threads (or processes) to a shared resource (critical section). It allows shared access for read-only use of the resource, while write operations access the resource exclusively. Such locks are ubiquitous in modern systems, and can be found, for example, in database software, file systems, key-value stores and operating systems.

Reader-writer locks have to keep track of the presence of active readers before a writer can be granted the lock. In the common case, such presence is recorded in a shared counter, incremented and decremented with every acquisition and release of the lock in the read mode. This is the way reader-writer locks are implemented in the Linux kernel, POSIX pthread library and several other designs [3, 35, 42]. The use of a shared counter lends itself to a relatively simple implementation and has a compact memory representation for a lock. However, it suffers under high intensity read-dominated workloads when the "reader indicator" state is updated frequently by a diverse set of threads, causing cache invalidation and coherence traffic [7, 17, 20, 31].

Alternative designs for reader-writer locks use distributed reader indicators, for instance, one per NUMA (non-uniform memory access) node as in cohort locks [6], or even one lock per core as in the Linux kernel brlock [10] and other related ideas [26, 31, 39, 46]. This improves reader-reader scalability, but also considerably increases the size of each lock instance. Furthermore, the lock performance is hampered when writes are frequent, as multiple indicators have to be accessed and/or modified. Finally, such locks have to be instantiated dynamically, since the number of sockets or cores can be unknown until the runtime. As a result, designers of modern reader-writer locks confront a difficult trade-off related to the scalability of maintaining the indication of the readers' presence.

In this paper, we propose a simple transformation, called BRAVO, that augments any existing reader-writer lock, adding just two integer fields to the lock instance. When applied on top of a counter-based reader-writer lock, BRAVO allows us to achieve, and often beat, the performance levels of locks that use distributed reader indicators while maintaining a compact footprint of the underlying lock. With BRAVO, readers make their presence known to writers by hashing their

thread's identity with the lock address, forming an index into a *visible readers table*. Readers attempt to install the lock address into the element (slot) in the table identified by that index. If successful, readers can proceed with their critical section *without* modifying the shared state of the underlying lock. Otherwise, readers resort to the acquisition path of the underlying lock. Note that the visible readers table is shared by all locks and threads in an address space. Crucially, readers of the same lock tend to write to different locations in the table, reducing coherence traffic and thus resulting in a NUMA-friendly design. At the same time, a writer always uses the acquisition path of the underlying lock, but also scans the readers table and waits for all readers that acquired that lock through it. A simple mechanism is put in place to limit the overhead of scanning the table for workloads in which writes are frequent.

We implemented BRAVO and evaluated it on top of several locks, such as the POSIX pthread_rwlock lock and the PF-Q reader-writer lock by Brandenburg and Anderson [3]. For our evaluation, we used numerous microbenchmarks as well as `rocksdb` [40], a popular open-source key-value store. Furthermore, we integrated BRAVO with `rwsem`, a read-write semaphore in the Linux kernel. We evaluated the modified kernel through kernel microbenchmarks as well as several user-space applications (from the Metis suite [33]) that create contention on read-write semaphores in the kernel. All our experiments in user-space and in the kernel demonstrate that BRAVO is highly efficient in improving performance and scalability of underlying locks in read-heavy workloads while introducing virtually no overhead, even in workloads in which writes are frequent.

The rest of the paper is organized as following. The related work is surveyed in Section 2. We present the BRAVO algorithm in Section 3 and discuss how we apply BRAVO in the Linux kernel in Section 4. The performance evaluation in user-space and the Linux kernel is provided in Sections 5 and 6, respectively. We conclude the paper and elaborate on multiple directions for future work in Section 7.

2 Related Work

► **Reader-Indicator Design** Readers that are *active* – currently executing in a reader critical section – must be visible to potential writers. Writers must be able to detect active readers in order to resolve read-vs-write conflicts, and wait for active readers to depart. The mechanism through which readers make themselves visible is the *reader indicator*. Myriad designs have been described in the literature. At one end of the spectrum we find a centralized reader indicator implemented as an integer field within each reader-writer lock instance that reflects the number of active readers. Readers use atomic instructions (or a central lock) to safely increment and decrement this field. Classic examples of such locks can be found in the early work of Mellor-Crummey and Scott [35] and more recent work by Shirako et al. [42]. Another reader-writer lock algorithm having a compact centralized reader indicator is Brandenburg and Anderson's Phase-Fair Ticket lock, designated `PF-T` in [3], where the reader indicator is encoded in two central fields. Their Phase-Fair Queue-based lock, `PF-Q`, uses a centralized counter for active readers and an MCS-like central queue, with local spinning, for readers that must wait. We refer to this latter algorithm as "BA" throughout the remainder of this paper. Such approaches are compact, having a small per-lock footprint, and simple, but, because of coherence traffic, do not scale in the presence of concurrent readers that are arriving and departing frequently [7, 17, 20, 31].

To address this concern, many designs turn toward distributed reader indicators. Each cohort reader-writer lock [6], for instance, uses a per-NUMA node reader indicator. While distributed reader indicators improve scalability, they also significantly increase the footprint of a lock instance, with each reader indicator residing on its own private cache line or sector to reduce false sharing. In addition, the size of the lock is variable with the number of nodes, and not known at compile-time, precluding simple static preallocation of locks. Writers are also burdened with the overhead of checking multiple reader indicators. Kashyap et al. [27] attempt to address some of those issues by maintaining a dynamic list of per-socket structures and expand the lock instance on-demand. However, this only helps if a lock is accessed by threads running on a subset of nodes.

At the extreme end of the spectrum we find lock designs with reader indicators assigned per-CPU or per-thread [10, 26, 31, 39, 46]. These designs promote read-read scaling, but have a large variable-sized footprint. They also favor readers in that writers must traverse and examine all the reader-indicators to resolve read-vs-write conflicts, possibly imposing a performance burden on writers. We note that there are a number of varieties of such distributed locks: a set of reader-indicators coupled with a central mutual exclusion lock for writer permission, as found in cohort locks [6]; sets of mutexes where readers must acquire one mutex and writers must acquire all mutexes, as found in Linux kernel brlocks [10]; or sets of reader-writer locks where readers must acquire read permission on one lock, and writers must acquire write permission on all locks. To reduce the impact on writers, which must visit all reader indicators, some designs use a tree of distributed counters where the root element contains a sum of the indicators within the subtrees [30].

Dice et al. [19] devised *read-write byte-locks* for use in the *TLRW* software transactional memory infrastructure. Briefly, read-write byte-locks are reader-writer locks augmented with an array of bytes, serving as reader indicators, where indices in the array are assigned to favored threads that are frequent readers. These threads can simply set and clear these reader indicators with normal store operations. The motivation for read-write byte-locks was to avoid atomic read-modify-write

instructions, which were particularly expensive on the system under test. The design, as described, is not NUMA-friendly as the byte array occupies a single cache line.

In addition to distributing or dispersing the counters, individual counters can themselves be further split into constituent `ingress` and `egress` fields to further reduce write sharing. Arriving readers increment the ingress field and departing readers increment the egress field. Cohort reader-writer locks use this approach [6].

BRAVO takes a different approach, opportunistically representing active readers in the shared global visible readers table. The table (array) is fixed in size and shared over all threads and locks within an address space. Each BRAVO lock has, in addition to the underlying reader-writer lock, a boolean flag that indicates if reader bias is currently enabled for that lock. Publication of active readers in the array is strictly optional and best-effort. A reader can always fall back to acquiring read permission via the underlying reader-writer lock. BRAVO's benefit comes from reduced coherence traffic arising from reader arrival. Such coherence traffic is particularly costly on NUMA systems, consuming shared interconnect bandwidth and also exhibiting high latency. As such, BRAVO is naturally NUMA-friendly. However, unlike most other NUMA-aware reader-writer locks, it does not need to understand or otherwise query the system topology, further simplifying the design and reducing dependencies[1].

► **Optimistic Invisible Readers** Synchronization constructs such as *seqlocks* [9,23,29] allow concurrent readers, but forgo the need for readers to make themselves visible. Critically, readers do not write to synchronization data and thus do not induce coherence traffic. Instead, writers update state – typically a modification counter – to indicate that updates have occurred. Readers check that counter at the start and then again at the end of their critical section, and if writers were active or the counter changed, the readers self-abort and retry. An additional challenge for seqlocks is that readers can observe inconsistent state, and special care must be taken to constrain the effects and avoid errant behavior in readers. Often, non-trivial reader critical sections must be modified to safely tolerate optimistic execution. Various hybrid forms exist, such as the `StampedLock` [36] facility in `java.util.concurrent`, which consists of a reader-writer lock coupled with a seqlock, providing 3 modes: classic pessimistic write locking, classic pessimistic read locking, and optimistic reading.

To avoid the problem where optimistic readers might see inconsistent state, transactional lock elision [16,18,24,28,38] based on hardware transactional memory can be used. Readers are invisible and do not write to shared data. Such approaches can be helpful, but are still vulnerable to indefinite abort and progress failure. In addition, the hardware transactional memory facilities required to support lock elision are not available on all systems, and are usually best-effort, without any guaranteed progress, requiring some type of fallback to pessimistic mechanisms.

► **Biased Locking** BRAVO draws inspiration from *biased locking* [14,22,37,41,44]. Briefly, biased locking allows the same thread to repeatedly acquire and release a mutual exclusion lock without requiring atomic instructions, except on the initial acquisition. If another thread attempts to acquire the lock, then expensive *revocation* is required to wrest bias from the original thread. The lock would then revert to normal non-biased mode for some period before again becoming potentially eligible for bias. (Conceptually, we can think of the lock as just being left in the locked state until there is contention. Subsequent lock and unlock operations by the original thread are ignored – the unlock operation is deferred until contention arises). Biased locking was a response to the CPU-local latencies incurred by atomic instructions on early Intel and SPARC processors and to the fact that locks in Java were often dominated by a single thread. Subsequently, processor designers have addressed the latency concern, rendering biased locking less profitable.

Classic biased locking identifies a preferred thread, while BRAVO identifies a preferred access mode. That is, BRAVO biases toward a mode instead of thread identity. BRAVO is suitable for read-dominated workloads, allowing a fast-path for readers when reader bias is enabled for a lock. If a write request is issued against a reader-biased lock, reader bias is disabled and revocation (scanning of the visible readers table) is required, shifting some cost from readers to writers. Classic biased locking provides benefit by reducing the number of atomic operations and improving latency. It does not improve scalability. BRAVO reader-bias, however, can improve both latency and scalability by reducing coherence traffic on the reader indicators in the underlying reader-writer lock.

3 The BRAVO Algorithm

BRAVO transforms any existing reader-writer lock *A* into *BRAVO-A*, which provides scalable reader acquisition. We say *A* is the *underlying* lock in *BRAVO-A*. In typical circumstances *A* might be a simple compact lock that suffers under high levels of reader concurrency. *BRAVO-A* will also be compact, but is NUMA-friendly as it reduces coherence traffic and offers scalability in the presence of frequently arriving concurrent readers.

Listing 1 depicts a pseudo-code implementation of the BRAVO algorithm. BRAVO extends *A*'s structure with a new `RBias` boolean field (Line 2). Arriving readers first check the `RBias` field, and, if found set, then hash the address of the lock with a value reflecting the calling thread's identity to

[1] While BRAVO is topology oblivious, it does require high-resolution low-latency means of reading the system clock. We further expect that reading the clock is scalable, and that concurrent readers do not interfere with each other. On systems with modern Intel CPUs and Linux kernels the `RDTSCP` instruction or `clock_gettime(CLOCK_MONOTONIC)` fast system call suffice.

form an index into the visible readers table (Lines 12–13). (This readers table is shared by all locks and threads in an address space. In all our experiments we sized the table at 4096 entries. Each table element, or *slot*, is either `null` or a pointer to a reader-writer lock instance). The reader then uses an atomic compare-and-swap (CAS) operator to attempt to change the element at that index from `null` to the address of the lock, publishing its existence to potential writers (Line 14). If the CAS is successful then the reader rechecks the `RBias` field to ensure it remains set (Line 18). If so, the reader has successfully gained read permission and can enter the critical section (Line 19). Upon completing the critical section the reader executes the complementary operation to release read permission, simply storing `null` into that slot (Lines 29–31). We refer to this as the *fast-path*. The fast-path attempt prefix (Lines 11-23) runs in constant time. Our hash function is based on the *Mix32* operator found in [43].

If the recheck operation above happens to fail, as would be the case if a writer intervened and cleared `RBias` and the reader lost the race, then the reader simply clears the slot (Line 21) and reverts to the traditional *slow-path* where it acquires read permission via the underlying lock (Line 24). Similarly, if the initial check of `RBias` found the flag clear (Line 12), or the CAS failed because of collisions in the array (Line 14) – the slot was found to be populated – then control diverts to the traditional slow-path. After a slow-path reader acquires read permission from the underlying lock, it enters and executes the critical section, and then at unlock time releases read permission via the underlying lock (Line 33).

Arriving writers first acquire write permission on the underlying reader-writer lock (Line 36). Having done so, they then check the `RBias` flag (Line 37). If set, the writer must perform *revocation*, first clearing the `RBias` flag (Line 40) and then scanning all the elements of the visible readers table checking for conflicting fast-path readers (Lines 42–44). If any elements match the lock, the writer must wait for that fast-path reader to depart and clear the slot. If lock L has 2 fast-path active readers, for instance, then L will appear twice in the array. Scanning the array might appear to be onerous, but in practice the sequential scan is assisted by the automatic hardware prefetchers present in modern CPUs. We observe a scan rate of about 1.1 nanoseconds per element on our system-under-test (described later). Having checked `RBias` and performed revocation if necessary, the writer then enters the critical section (Line 50). At unlock-time, the writer simply releases write permission on the underlying reader-writer lock (Line 51). Therefore the only difference for writers under BRAVO is the requirement to check and potentially revoke reader bias if `RBias` was found set.

We note that writers only scan the visible reader table, and never write into it. Yet, this scan may pollute the writer's cache. One way to cope with it is to use non-temporal loads, however, exploring this idea is left for the future work. Note that revocation is only required on transitions from reading to

```
1  class BRAVOLock<T> :
2    int RBias
3    Time InhibitUntil
4    T Underlying
5
6  ## Shared global :
7  BRAVOLock * VisibleReaders [4096]
8  int N = 9   # slow-down guard
9
10 def Reader(BRAVOLock * L) :
11   BRAVOLock * * slot = null
12   if L.RBias :
13     slot = VisibleReaders + Hash(L, Self)
14     if CAS(slot, null, L) == null :
15       # CAS succeeded
16       # store-load fence required on TSO
17       # typically subsumed by CAS
18       if L.RBias :          # recheck
19         goto EnterCS        # fast path
20       *slot = null          # raced - RBias changed
21     slot = null
22   # Slow path
23   assert slot == null
24   AcquireRead (L.Underlying)
25   if L.RBias == 0 and Time() >= L.InhibitUntil :
26     L.RBias = 1
27   EnterCS:
28   ReaderCriticalSection()
29   if slot != null :
30     assert *slot == L
31     *slot = null
32   else :
33     ReleaseRead (L.Underlying)
34
35 def Writer(BRAVOLock * L) :
36   AcquireWrite (L.Underlying)
37   if L.RBias :
38     # revoke bias
39     # store-load fence required on TSO
40     L.RBias = 0
41     auto start = Time()
42     for i in xrange(VisibleReaders) :
43       while VisibleReaders[i] == L :
44         Pause()
45     auto now = Time()
46     # primum non-nocere :
47     # limit and bound slow-down
48     # arising from revocation overheads
49     L.InhibitUntil = now + ((now - start) * N)
50   WriterCriticalSection()
51   ReleaseWrite (L.Underlying)
```

Listing 1: Simplified Python-like implementation of BRAVO

writing and only when `RBias` was previously set.

In summary, active readers can make their existence public in one of two ways : either via the visible readers table (fast-path), or via the traditional underlying reader-writer lock (slow-path). Our mechanism allows both slow-path and fast-path readers simultaneously. Absent hash collisions, concurrent fast-path readers will write to different locations in the visible readers table. Collisions are benign, and impact per-

formance but not correctness. Writers resolve read-vs-write conflicts against fast-path readers via the visible readers table and against slow-path readers via the underlying reader-writer lock.

One important remaining question is how to set `RBias`. In our early prototypes we set `RBias` in the reader slow-path based on a low-cost Bernoulli trial with probability $P = 1/100$ using a thread-local Marsgalia XOR-Shift [34] pseudo-random number generator. While this simplistic policy for enabling bias worked well in practice, we were concerned about situations where we might have enabled bias too eagerly, and incur frequent revocation to the point where *BRAVO-A* might be slower than *A*. Specifically, the worst-case scenario would be where slow readers repeatedly set `RBias`, only to have it revoked immediately by a writer.

The key additional cost in BRAVO is the revocation step, which executes under the underlying write lock and thus serializes operations associated with the lock[2]. As such, we measure the latency of revocation and multiply that period by N, a configurable parameter, and then inhibit the subsequent setting of bias in the reader slow-path for that period, bounding the worst-case expected slow-down from BRAVO for writers to $1/(N+1)$ (cf. Lines 41-49). Our specific performance goal is *primum non nocere* – first, do no harm, with *BRAVO-A* never underperforming *A* by any significant margin on any workload[3]. This tactic is simple and effective, but excessively conservative, taking into account only the worst-case performance penalty imposed by BRAVO, and not accounting for any potential benefit conferred by the BRAVO fast-path. Furthermore, measuring the revocation duration also incorporates the waiting time, as well as the scanning time, yielding a conservative over-estimate of the revocation scan cost and resulting in less aggressive use of reader bias. Despite these concerns, we find this policy yields good and predictable performance. For all benchmarks in this paper we used $N = 9$ yielding a worst-case writer slow-down bound of about 10%. Our policy required adding a second BRAVO-specific timestamp field `InhibitUntil` (Line 3), which reflects the earliest time at which slow readers should reenable bias[4]. We note that for safety, readers can only set `RBias` while they hold read permission on the underlying reader-writer lock, avoiding interactions with writers (cf. Lines 25–26).

In our implementations revoking waiters busy-wait for readers to depart. There can be at most one such busy-waiting thread for a given lock at any given time. We note, however, that it is trivial to shift to a waiting policy that uses blocking.

BRAVO acts as an accelerator layer, as readers can always fall back to the traditional underlying lock to gain read access. The benefit arises from avoiding coherence traffic on the centralized reader indicators in the underlying lock, and instead relying on updates to be diffused over the visible readers table. Fast-path readers write only into the visible readers table, and not the lock instance proper. This access pattern improves performance on NUMA systems, where write sharing is particularly expensive. We note that if the underlying lock algorithm *A* has reader preference or writer preference, then *BRAVO-A* will exhibit that same property. Write performance and the scalability of read-vs-write and write-vs-write behavior depends solely on the underlying lock. Under high write intensity, with write-vs-write and write-vs-read conflicts, the performance of BRAVO devolves to that of the underlying lock. BRAVO accelerates reads only. BRAVO fully supports the case where a thread holds multiple locks at the same time.

BRAVO supports *try-lock* operations as follows. For read *try-lock* attempts an implementation could try the BRAVO fast path and then fall back, if the fast path fails, to the slow path underling *try-lock*. An implementation can also opt to forgo the fast path attempt and simply call the underlying *try-lock* operator. We use the former approach when applying BRAVO in the Linux kernel as detailed in the next section. We note that if the underlying *try-lock* call is successful, one may set `RBias` if the BRAVO policy allows that (e.g., if the current time is larger than `InhibitUntil`). For write *try-lock* operators, an implementation will invoke the underlying *try-lock* operation. If successful, and bias is set, then revocation must be performed following the same procedure described in Lines 37–49.

As seen in Listing 1, the *slot* value must be passed from the read lock operator to the corresponding unlock. `null` indicates that the slow path was used to acquire read permission. To provide correct errno values in the POSIX pthread environment, a thread must be able to determine if it holds read, write, or no permission on a given lock. This is typically accomplished by using per-thread lists of locks currently held in read mode. We leverage those list elements to pass the slot. We note that the Cohort read-write lock implementation [6] passed the reader's NUMA node ID from lock to corresponding unlock in this exact fashion.

4 Applying BRAVO to the Linux Kernel `rwsem`

In this section, we describe prototype integration of BRAVO in the Linux kernel, where we apply it to `rwsem`. `Rwsem` is a read-write semaphore construct. Among many places inside the kernel, it is used to protect the access to the virtual memory area (VMA) structure of each process [11], which makes it a

[2]Additional costs associated with BRAVO include futile atomic operations from collisions, and sharing or false-sharing arising from near-collisions in the table. Our simplified cost model ignores these secondary factors. We note that the odds of collision are equivalent to those given by the "Birthday Paradox" [48] and that the general problem of deciding to set bias is equivalent to the classic "ski-rental" problem [47].

[3]Our approach conservatively forgoes the potential of better performance afforded by the aggressive use of reader bias in order to limit the possibility of worsened performance [49].

[4]We observe that it is trivial to collapse `RBias` and `InhibitUntil` into just a single field. For clarity, we did not do so in our implementation.

source of contention for data intensive applications [8, 11].

On a high level, rwsem consists of a counter and a waiting queue protected by a spin-lock. The counter keeps track of the number of active readers, as well as encodes the presence of a writer. To acquire the rwsem in the read mode, a reader atomically increments the counter and checks its value. If a (waiting or active) writer is not present, the read acquisition is successful; otherwise, the reader acquires the spin-lock protecting the waiting queue, joins the queue at the tail, releases the spin-lock and blocks, waiting for a wake-up signal from a writer. As a result, when there is no reader-writer contention, the read acquisition boils down to one atomic counter increment. On architectures that do not support an atomic increment instruction, this requires acquisition (and subsequent release) of the spin-lock. Even on architectures that have such an instruction (such as Intel x86), the read acquisition of rwsem creates contention over the cache line hosting the counter.

In our integration of BRAVO on top of rwsem, we make a simplifying assumption that the semaphore is always released by the same thread that acquired it for read. This is not guaranteed by the API of rwsem, however, this is a common way of using semaphores in the kernel. This assumption allows us to preserve the existing rwsem API and limits the scope of changes required, resulting in a patch of only three files and adding just a few dozens lines of code. We use this assumption when determining the slot into which a thread would store the semaphore address on the fast acquisition path, and clear that slot during the release operation[5].

While we have not observed any issue when running and evaluating the modified kernel, we note that our assumption can be easily eliminated by, for example, extending the API of rwsem to allow an additional pointer argument for read acquisition and release functions. In case the acquisition is made on the fast path, this pointer would be used to store the address of the corresponding slot; later, this pointer can be passed to a (different) releasing thread to specify the slot to be cleared. Alternatively, we can extend the API of rwsem to include a flag explicitly allowing the use of the fast path for read acquisition and release. This flag would be set only in call sites known for high read contention (such as in functions operating on VMAs), where a thread that releases the semaphore is known to be the one that acquired it. Other call sites for semaphore acquisition and release can be left untouched, letting them use the slow path only.

We note that the default configuration of the kernel enables a so-called spin-on-owner optimization of rwsem [32]. With this optimization, the rwsem structure includes an owner field that contains a pointer to the current struct of the owner task when rwsem is acquired for write. Using this field, a reader may check whether the writer is currently running on a CPU, and if so, spin rather than block [32]. While writers do not use this field to decide whether they have to spin (as there might be multiple readers), in the current rwsem implementation a reader updates the owner field regardless, storing there its current pointer along with a few control bits (that specify that the lock is owned by a reader). These writes by readers are for debugging purposes only, yet they create unnecessary contention on the owner field. We fix that by letting a reader set only the control bits in the owner field, and only if those bits were not set before, i.e., when the first reader acquires that rwsem instance after a writer. Note that all subsequent readers would read, but not update the owner field, until it is updated again by a writer.

[5]We determine the slot by hashing the task struct pointer (current) with the address of the semaphore.

5 User-space Evaluation

All user-space data was collected on an Oracle X5-2 system. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total. The system was running Ubuntu 18.04 with a stock Linux version 4.15 kernel, and all software was compiled using the provided GCC version 7.3 toolchain at optimization level "-O3". 64-bit code was used for all experiments. Factory-provided system defaults were used in all cases, and *turbo mode* [45] was left enabled. In all cases default free-range unbound threads were used.

We implemented all locks within LD_PRELOAD interposition libraries that expose the standard POSIX pthread_rwlock_t programming interface. This allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses reader-writer locks. This same framework was used to implement Cohort reader-writer locks [6].

In the following figures "BA" refers to the Brandenburg-Anderson PF-Q lock [3]; "Cohort-RW" refers to the C-RW-WP lock [6]; "Per-CPU" reflects a lock that consists of an array of BA locks, one for each CPU, where readers acquire read-permission on the sub-lock associated with their CPU, and writers acquire writer permission on all the sub-locks (this lock is similar to the Linux kernel brlock construct [10]); "pthread" is a default Linux POSIX "pthread_rwlock" read-write lock mechanism; "BRAVO-BA" reflects BRAVO implemented on top of BA and "BRAVO-pthread" is BRAVO implemented on top of pthread_rwlock.

We also experimented with several other reader-writer locks. In particular, we took data on the Brandenburg-Anderson PF-T lock and the BRAVO form thereof. PF-T implements the reader indicator via a central pair of counters, one incremented by arriving readers and the other incremented by departing readers. Waiting readers busy-wait on a dedicated *writer present* bit encoded in the reader arrival counter. In PF-Q active readers are tallied on a central pair of counters in the same fashion as PF-T, but waiting readers enqueue on an MCS-like queue. In both PF-T and PF-Q, arriving read-

ers update the central reader indicator state, generating more coherence traffic than would be the case for locks that use distributed reader indicators or BRAVO. Waiting readers in PF-T use global spinning, while waiting readers in PF-Q use local spinning on a thread-local field in the enqueued element. PF-T enjoys slightly shorter code paths but also suffers from lessened scalability because of the global spinning. We found that PF-T and PF-Q offer broadly similar performance, with PF-T having a slight advantage when the arrival rate is high, the number of waiting threads is low, and the waiting period is shorter. PF-T is slightly more compact having just 4 integer fields, while PF-Q has 2 such fields and 4 pointers. For brevity, we do not include PF-T results. We also found that "fair lock with local only spinning" by Mellor-Crummey and Scott [35] yielded performance similar to or slower than that of PF-Q.

We note that the default pthread read-write lock implementation found in our Linux distribution provides strong reader preference, and admits indefinite writer starvation[6]. The reader indicator is centralized and the lock has a footprint of 56 bytes for 64-bit programs. Waiting threads block immediately in the kernel without spinning. While this policy incurs overheads associated with voluntary context switching, it may also yield benefits by allowing "polite" waiting by enabling turbo mode for those threads making progress. Except where otherwise noted, we plot the number of concurrent threads on the X-axis, and aggregate throughput on the Y-axis, and report the median of 7 independent runs for each data point.

We use a 128 byte sector size on Intel processors for alignment to avoid false sharing. The unit of coherence is 64 bytes throughout the cache hierarchy, but 128 bytes is required because of the adjacent cache line prefetch facility where pairs of lines are automatically fetched together. BA requires just 128 bytes – 2 32-bit integer fields plus 4 pointers fields with the overall size rounded up to the next sector boundary. BRAVO-BA adds the 8-byte `InhibitUntil` field, which contains a timestamp, and the 4-byte `RBias` field. Rounding up to the sector size, this still yields a 128 byte lock instance. Per-CPU consists of one instance of BA for each logical CPU, yielding a lock size of 9216 bytes on our 72-way system. Cohort-RW consists of one reader indicator (128 bytes) per NUMA node, a central location for state (128 bytes) and a full cohort mutex [21] to provide writer exclusion. In turn, the cohort mutex requires one 128-byte sub-lock per NUMA node, and another 256 bytes for central state. Thus, the total size of the Cohort-RW lock on our dual-socket system is 896 bytes. (While our implementation did not do so, we note that a more space aggressive implementation of Cohort-RW could colocate the per-node reader indicators with the mutex sub-locks, and the central state for the read-write lock with its associated cohort mutex, yielding a size of 512 bytes).

[6] The pthread implementation allows writer preference to be selected via a non-portable API. Unfortunately this feature currently has bugs that result in lost wakeups and hangs: https://sourceware.org/bugzilla/show_bug.cgi?id=23861.

As noted above, the size of the pthread read-write lock is 56 bytes, and the BRAVO variant adds 12 bytes. The size of BA, BRAVO-BA, pthread, and BRAVO-pthread are fixed, and known at compile-time, while the size of Per-CPU varies with the number of logical CPUs, and the size of Cohort-RW varies with the number of NUMA nodes. Finally, we observe that BRAVO allows more relaxed approach toward the alignment and padding of the underlying lock. Since fast-path readers do not mutate the underlying lock fields, the designer can reasonably forgo alignment and padding on that lock, without trading off reader scalability.

The size of the lock can be important in concurrent data structures, such as linked lists or binary search trees, that use a lock per node or entry [4, 12, 25]. As Bronson at el. observe, when a scalable lock is striped across multiple cache lines to avoid contention in the coherence fabric, it is "prohibitively expensive to store a separate lock per node" [4].

BRAVO also requires the visible readers table. With 4096 entries on a system with 64-bit pointers, the additional footprint is 32KB. The table is aligned and sized to minimize the number of underlying pages (reducing TLB footprint) and to eliminate false sharing from variables that might be placed adjacent to the table. We selected a table size 4096 empirically but in general believe the size should be a function of the number of logical CPUs in the system. Similar tables in the linux kernel, such as the *futex* hash table, are sized in this fashion [5].

BRAVO yields a favorable performance trade-off between space and scalability, offering a viable alternative that resides on the design spectrum between classic centralized locks, such as BA, having small footprint and poor reader scalability, and the large locks with high reader scalability.

5.1 Sensitivity to Inter-Lock Interference

As the visible readers array is shared over all locks and threads within an address space, one potential concern is collisions that might arise when multiple threads are using a large set of locks. Near collisions are also of concern as they can cause false sharing within the array. To determine BRAVO's performance sensitivity to such effects, we implemented a microbenchmark program that spawns 64 concurrent threads. Each thread loops as follows: randomly pick a reader-writer lock from a pool of such locks; acquire that lock for read; advance a thread-local pseudo-random number generator 20 steps; release read permission on the lock; and finally advance that random number generator 100 steps. At the end of a 10 second measurement interval we report the number of lock acquisitions. No locks are ever acquired with write permission. Each data point is the median of 7 distinct runs. We report the results in Figure 1 where the X-axis reflects the number of locks in the pool (varying through powers-of-two between 1 and 8192) and the Y-axis is the number of acquisitions completed by BRAVO-BA divided by the number completed by a

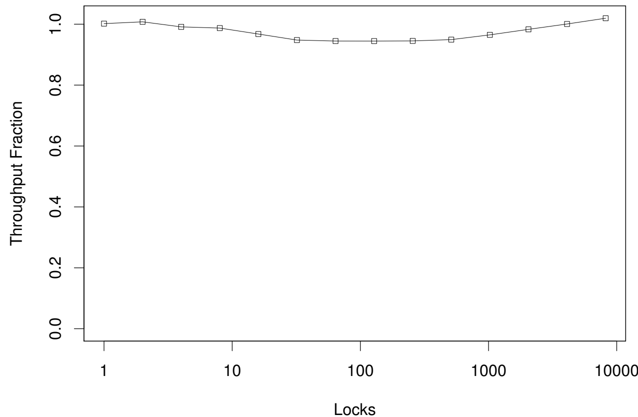


Figure 1: Inter-lock interference

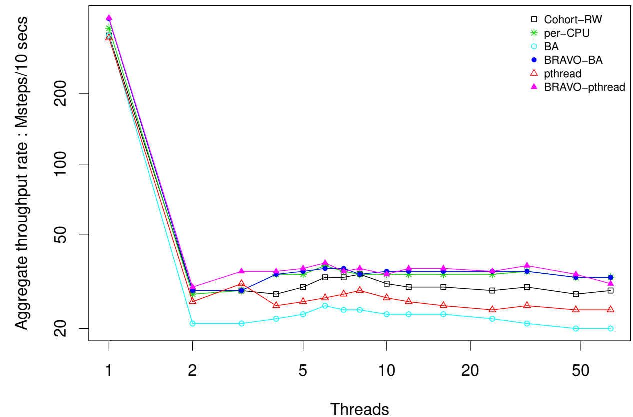


Figure 2: Alternator

specialized version of BRAVO-BA where each lock instance has a private array of 4096 elements. This fraction reflects the performance drop attributable to inter-lock conflicts and near conflicts in the shared array, where the modified form of BRAVO-BA can be seen as an idealized form that has a large per-instance footprint but which is immune to inter-lock conflicts[7]. The worst-case penalty arising from inter-lock interference (the lowest fraction value) is always under 6%.

5.2 Alternator

Figure 2 shows the results of our `alternator` benchmark. The benchmark spawns the specified number of concurrent threads, which organize themselves into a logical ring, each waiting for notification from its "left" sibling. Notification is accomplished via setting a thread-specific variable with a store instruction and waiting is via simple busy-waiting. Once notified, the thread acquires and then immediately releases read permission on a shared reader-writer lock. Next the thread notifies its "right" sibling and then again waits. There are no writers, and there is no concurrency between readers. At most one reader is active at any given moment. At the end of a 10 second measurement interval the program reports the number of notifications.

The BA lock suffers as the lines underlying the reader indicators "slosh" and migrate from cache to cache. In contrast BRAVO-BA readers touch different locations in the visible readers table as they acquire and release read permissions. BRAVO enables reader-bias early in the run, and it remains set for the duration of the measurement interval. All locks experience a significant performance drop between 1 and 2 threads due to the impact of coherent communication for notification. Crucially, we see that BRAVO-BA outperforms the underlying BA by a wide margin, and is competitive with the much larger Per-CPU lock. In addition, the performance of BA can be seen to degrade as we add threads, whereas the performance of BRAVO-BA remains stable. The same observations are true when considering BRAVO-pthread and pthread locks.

Since the hash function that associates a read locking request with an index is deterministic, threads repeatedly locking and unlocking a specific lock will enjoy temporal locality and reuse in the visible readers table.

5.3 test_rwlock

We next report results from the `test_rwlock` benchmark described by Desnoyers et al. [13][8]. The benchmark was designed to evaluate the performance and scalability of reader-writer locks against the RCU (Read-Copy Update) synchronization mechanism. We used the following command-line: `test_rwlock T 1 10 -c 10 -e 10 -d 1000`. The benchmark launches 1 fixed-role writer thread and T fixed-role reader threads for a 10 second measurement interval. The writer loops as follows: acquire a central reader-writer lock instance; execute 10 units of work, which entails counting down a local variable; release writer permission; execute a non-critical section for 1000 work units. Readers loop acquiring the central lock for reading, executing 10 steps of work in the critical section, and then release the lock. (The benchmark has no facilities to allow a non-trivial critical section for readers). At the end of the measurement interval the benchmark reports the sum of iterations completed by all the threads. As we can see in Figure 3, BRAVO-BA significantly outperforms BA, and even the Cohort-RW lock at higher thread counts. Since the workload is extremely read-dominated, the Per-CPU lock yields the best performance, albeit with a very

[7]We note that as we increase the number of locks, cache pressure constitutes a confounding factor for the specialized version of BRAVO-BA. For full discussion, see the extended version of this paper available at https://arxiv.org/abs/1810.01553.

[8]obtained from https://github.com/urcu/userspace-rcu/blob/master/tests/benchmark/test_rwlock.c and modified slightly to allow a fixed measurement interval.

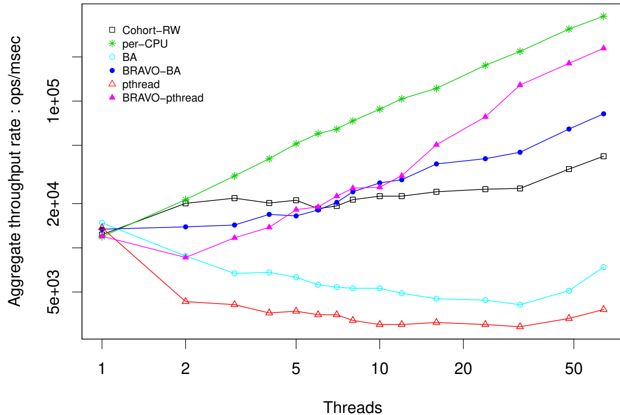


Figure 3: test_rwlock

large footprint and only because of the relatively low write rate. For that same reason, and due to its default reader preference, BRAVO-pthread easily beats pthread, and comes close to the performance level of Per-CPU.

5.4 RWBench

Using `RWBench` – modeled on a benchmark of the same name described by Calciu et al. [6] – we evaluated the reader-write lock algorithms over a variety of read-write ratios, ranging from write-intensive in Figure 4a (9 out of every 10 operations are writes) to read-intensive in Figure 4f (1 out of every 10000 operations are writes), demonstrating that BRAVO inflicts no harm for write-intensive workloads, but improves performance for more read-dominated workloads. `RWBench` launches T concurrent threads for a 10 second measurement interval. Each thread loops as follows: using a thread-local pseudo-random generator, decide to write with probability P via a Bernoulli trial; writers acquire a central reader-write lock for write permission and then execute 10 steps of a thread-local C++ `std::mt19937` random number generator and then release write permission, while readers do the same, but under read permission; execute a non-critical section of N steps of the same random-number generator where N is a random number uniformly distributed in $[0,200)$ with average and median of 100. At the end of the measurement interval the benchmark reports the total number of top-level loops completed.

In Figure 4a we see poor scalability over all the locks by virtue of the highly serialized write-heavy nature of the workload. Per-CPU fairs poorly as writes, which are common, need to scan the array of per-CPU sub-locks. Cohort-RW provides some benefit, while BRAVO-BA (BRAVO-pthread) tracks closely to BA (pthread, respectively), providing neither benefit nor harm. The same behavior plays out in Figure 4b ($P = 1/2$) and Figure 4c ($P = 1/10$), although in the latter we see some scaling from Cohort-RW. In Figure 4d ($P = 1/100$) we begin to see BRAVO-BA outperforming BA at higher thread counts. Figure 4e ($P = 1/1000$) and Figure 4f ($P = 1/10000$ – extremely read-dominated) are fairly similar, with BRAVO-BA and BRAVO-pthread yielding performance similar to that of Per-CPU, Cohort-RW yielding modest scalability, and BA and pthread yielding flat performance as the thread count increases.

5.5 `rocksdb` readwhilewriting

We next explore performance sensitivity to the reader-writer lock in the `rocksdb` database [40]. We observed high frequency reader traffic arising in the readwhilewriting benchmark from calls in `::Get()` to `GetLock()` defined in `db/memtable.cc`[9]. In Figure 5 we see the performance of BRAVO-BA and BRAVO-pthread tracks that of Per-CPU and always exceeds that of Cohort-RW and the respective underlying locks.

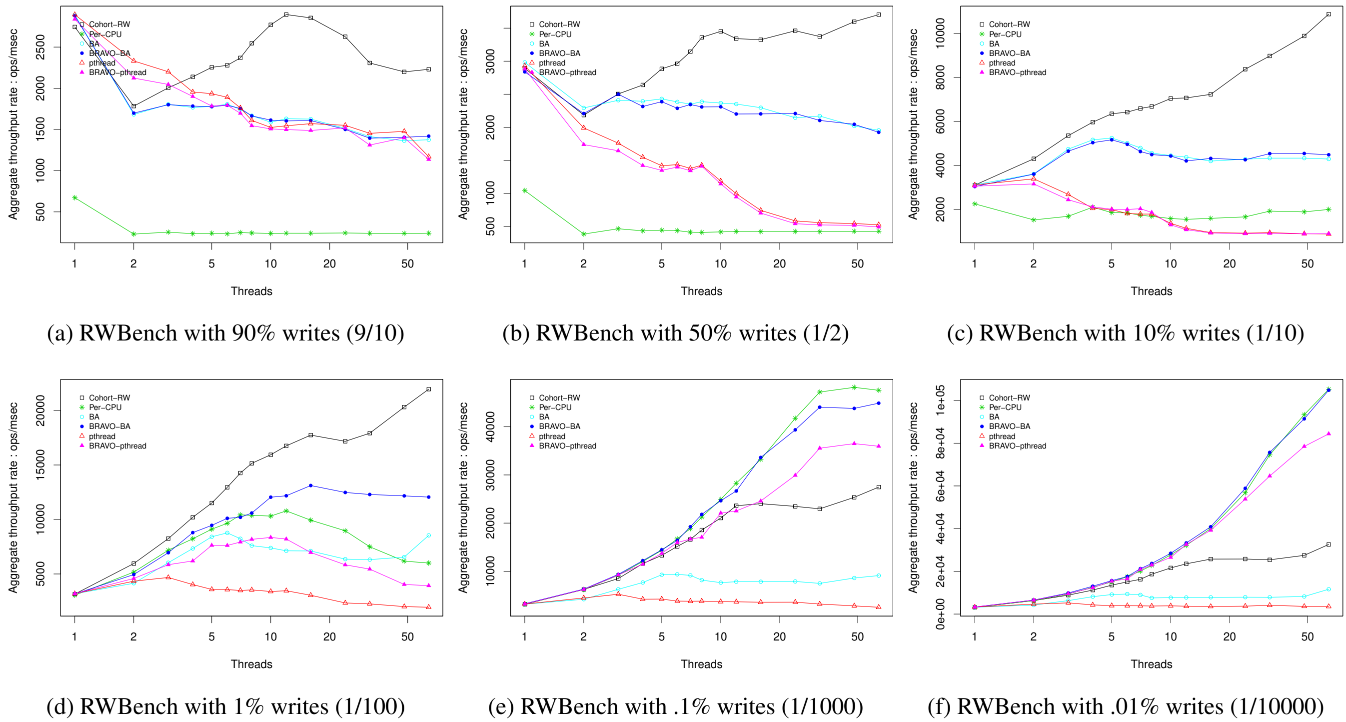
5.6 `rocksdb` hash_table_bench

`rocksdb` also provides a benchmark to stress the hash table used by their persistent cache[10]. The benchmark implements a central shared hash table as a C++ `std::unordered_map` protected by a reader-writer lock. The cache is pre-populated before the measurement interval. At the end of the 50 second measurement interval the benchmark reports the aggregate operation rate – reads, erases, insertions – per millisecond. A single dedicated thread loops, erasing random elements, and another dedicated thread loops inserting new elements with a random key. Both erase and insertion operations require write access. The benchmark launches T reader threads, which loop, running lookups on randomly selected keys. We vary T on the X-axis. All the threads execute operations back-to-back without a delay between operations. The benchmark makes frequent use of malloc-free operations in the `std::unordered_map`. The default malloc allocator fails to fully scale in this environment and masks any benefit conferred by improved reader-writer locks, so we instead used the index-aware allocator by Afek el al. [1].

The results are shown in Figure 6. Once again, BRAVO enhances the performance of underlying locks, and shows substantial speedup at high thread counts.

[9] We used `rocksdb` version 5.13.4 with the following command line: `db_bench -threads=T -benchmarks=readwhilewriting -memtablerep=cuckoo -duration=100 -inplace_update_support=1 -allow_concurrent_memtable_write=0 -num=10000 -inplace_update_num_locks=1 -histogram -stats_interval=10000000`

[10] https://github.com/facebook/rocksdb/blob/master/utilities/persistent_cache/hash_table_bench.cc run with the following command-line: `hash_table_bench -nread_thread=T -nsec=50`



(a) RWBench with 90% writes (9/10)

(b) RWBench with 50% writes (1/2)

(c) RWBench with 10% writes (1/10)

(d) RWBench with 1% writes (1/100)

(e) RWBench with .1% writes (1/1000)

(f) RWBench with .01% writes (1/10000)

Figure 4: RWBench

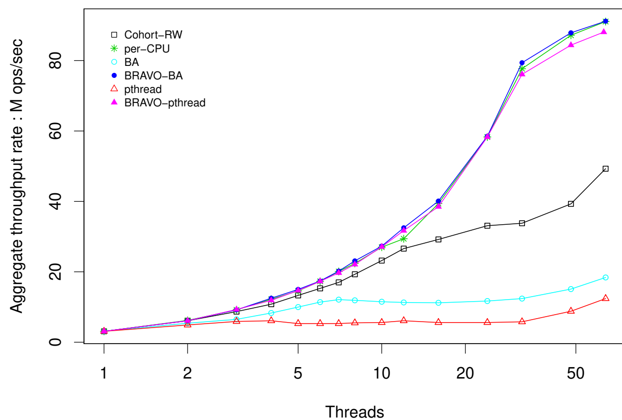


Figure 5: `rocksdb` readwhilewriting

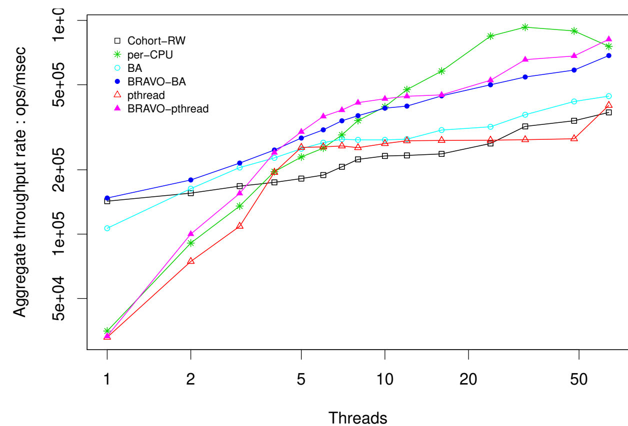


Figure 6: `rocksdb` hash_table_bench with `std::unordered_map`

6 Linux Kernel Experiments

All kernel-space data was collected on an Oracle X5-4 system. The system has 4 sockets, each populated with an Intel Xeon CPU E7-8895 v3 running at 2.60GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 144 logical CPUs in total. The patch was applied on top of a recent Linux version 4.20.rc4 kernel, which we refer to as *stock*. We refer to the kernel version modified to use BRAVO simply as *BRAVO*.

Factory-provided system defaults were used in all cases. In particular, we compiled the kernel in the default configuration, which notably disables the lock performance data collection mechanism (aka `lockstat`[11]) built into the kernel for debugging lock performance. As we mention below, this mechanism was useful to gain insights into usage patterns of kernel locks in various applications. However, we kept it disabled during

[11] https://www.kernel.org/doc/Documentation/locking/lockstat.txt

performance measurements as it adds a probing effect by generating stores into shared variables, e.g., by keeping track of the last CPU on which a given lock instance, `rwsem` included, was acquired. These stores hamper the benefit of techniques like BRAVO that aim to reduce updates to the shared state during lock acquisition and release operations.

Each experiment is repeated 7 times, and the reported numbers are the average of the corresponding results. Unless noted, the reported results were relatively stable, with variance of less than 5% from the average in most cases.

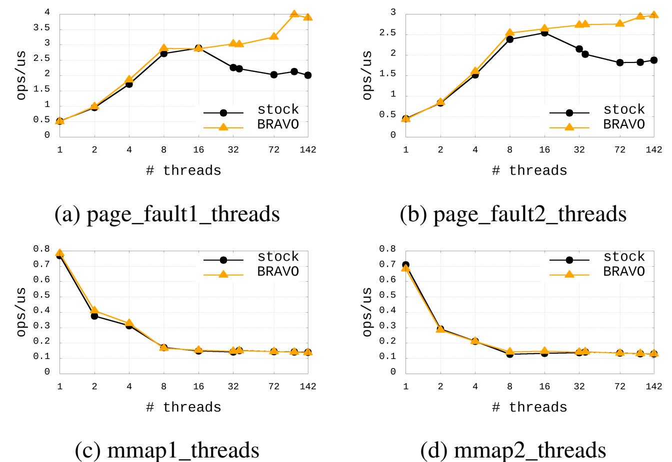
6.1 will-it-scale

`will-it-scale` is an open-source collection of microbenchmarks[12] for stress-testing various kernel subsystems. `will-it-scale` runs in user-mode but is known to induce contention on kernel locks [15]. Each microbenchmark runs a given number of tasks (that can be either threads or processes), performing a series of specific system calls (such as opening and closing a file, mapping and unmapping memory pages, raising a signal, etc.). We experiment with a subset of microbenchmarks that access the VMA structure and create contention on `mmap_sem`, an instance of `rwsem` that protects the access to VMA [11]. In particular, the relevant microbenchmarks are `page_fault` and `mmap`. The former continuously maps a large (128M) chunk of memory, writes one word into every page in the chunk (causing a page fault for every write), and unmaps the memory. The latter simply maps and unmaps large chunks of memory. (Each of those benchmarks has several variants denoted as `page_fault1`, `page_fault2`, etc.)

Page faults require the acquisition of `mmap_sem` for read, while memory mapping and unmapping operations acquire `mmap_sems` for write [8]. Therefore, the access pattern for `mmap_sem` is expected to be read-heavy in the `page_fault` microbenchmark and more write-heavy in `mmap`. We confirmed that through `lockstat` statistics. We note that BRAVO is not expected to provide any benefit for `mmap`, yet we include it to evaluate any overhead BRAVO might introduce in write-heavy workloads.

Figure 7 presents the results of our experiments for `page_fault` and `mmap`, respectively. In `page_fault`, the BRAVO version performs similarly to stock as long as the latter scales. After 16 threads, however, the throughput of the stock version decreases while the BRAVO version continues to scale, albeit at a slower rate. At 142 threads, BRAVO outperforms stock by up to 93%. At the same time, `mmap` shows no significant difference in the performance of BRAVO vs. stock, suggesting that BRAVO does not introduce overhead in scenarios where it is not profitable.

[12] https://github.com/antonblanchard/will-it-scale



(a) page_fault1_threads (b) page_fault2_threads

(c) mmap1_threads (d) mmap2_threads

Figure 7: `will-it-scale` results

6.2 Metis

Metis is an open-source MapReduce library [33] used in the past to assess the scalability of Linux kernel locks [2, 8, 27]. Metis is known for a relatively intense access to VMA through the mix of page-fault and mmap operations [27]. By collecting lock performance statistics with `lockstat`, however, we found that only few of Metis benchmarks have both a large number of `mmap_sem` acquisitions and a large portion of those acquisitions is for read. We note that like in all other kernel benchmarks, `lockstat` was disabled when measuring performance numbers reported below.

Tables 1 and 2 present the performance results, respectively, for `wc`, a map-reduce based word count, and `wrmem`, which allocates a large chunk of memory and fills it with random "words", which are fed into the map-reduce framework for inverted index calculation. The BRAVO version can achieve speedups of over 30%. We note that some of the data, particularly for `wc`, was noisy; we print values with variance larger than 5% from the mean in italics. (All values have variance of 19% or less). We also note that BRAVO did not create significant overhead for any other Metis benchmark, although some benchmarks produced noisy results similarly to `wc`.

7 Conclusion and Future Work

BRAVO easily composes with existing locks, preserving desirable properties of those underlying locks, and yielding a composite lock with improved read-read scalability. We specifically target read-dominated workloads with multiple concurrent threads that acquire and release read permission at a high rate. The approach is simple, effective, and yields improved performance for read-dominated workloads compared to commonly used compact locks. The key trade-off inherent in the design is the benefit accrued by reads against the potential slow-down imposed by revocation. Even in mixed or

#threads	stock	BRAVO	speedup
1	*18.059*	17.957	0.6%
2	12.189	12.090	0.8%
4	10.045	*9.652*	3.9%
8	*8.880*	*7.976*	10.2%
16	*13.321*	11.325	15.0%
32	23.654	*19.281*	18.5%
72	*119.018*	98.054	17.6%
108	132.147	111.139	15.9%
142	143.217	125.160	12.6%

Table 1: `wc` runtime (sec)

#threads	stock	BRAVO	speedup
1	279.553	279.423	0.0%
2	137.271	136.706	0.4%
4	68.989	69.013	0.0%
8	36.161	36.210	-0.1%
16	22.647	21.985	2.9%
32	18.332	14.707	19.8%
72	*48.397*	31.868	34.2%
108	58.529	*36.910*	36.9%
142	58.994	42.544	27.9%

Table 2: `wrmem` runtime (sec)

write-heavy workloads, we limit any slow-down stemming from revocation costs and bound harm, making the decision to use BRAVO simple. BRAVO incurs a very small footprint increase per lock instance, and also adds a shared table of fixed size that can be used by all threads and locks. BRAVO's key benefit arises from reducing coherence cost that would normally be incurred by locks having a central reader indicator. Write performance is left unchanged relative to the underlying lock. BRAVO provides read-read performance at, and often above, that of the best modern reader-writer locks that use distributed read indicators, but without the footprint or complexity of such locks. By reducing coherence traffic, BRAVO is implicitly NUMA-friendly.

► **Future directions** We identify a number of future directions for our investigation into BRAVO-based designs:

- Dynamic sizing of the visible readers table based on collisions. Large tables will have reduced collision rates, but larger scan revocation overheads.
- The reader fast-path currently probes just a single location and reverts to the slow-path after a collision. We plan on using a secondary hash to probe an alternative location. In that vein, we note that while we currently use a hash function to map a thread's identity and the lock address to an index in the table, there is no particular requirement that the function that associates a read request with an index be deterministic. We plan on exploring other functions, using time or random numbers to form indices. While this will be less beneficial in terms of cache locality for the reader, it might be helpful in case of temporal contention over specific slots.
- Accelerate the revocation scan operation via SIMD instructions such as AVX. The visible readers table is usually sparsely populated, making it amenable to such optimizations. As already noted, non-temporal non-polluting loads may also be helpful for the scan operation.
- As noted, our current policy to enable bias is conservative, and leaves untapped performance. We intend to explore more sophisticated adaptive policies based on recent behavior and to use a more faithful cost model.
- An interesting variation is to implement BRAVO on top of an underlying mutex instead of a reader-writer lock. Slow-path readers must acquire the mutex, and the sole source of read-read concurrency is via the fast path. We note that some applications might expect the reader-write lock implementation to be fully work conserving and *maximally admissive* – always allowing full read concurrency where available. For example, an active reader thread $T1$, understanding by virtue of application invariants that no writers are present, might signal another thread $T2$ and expect that $T2$ can enter a reader critical section while $T1$ remains within the critical section. This progress assumption would not necessarily hold if readers are forced through the slow path and read-read parallelism is denied.
- In our current implementation arriving readers are blocked while a revocation scan is in progress. This could be avoided by adding a mutex to each BRAVO-enhanced lock. Arriving writers immediately acquire this mutex, which resolves all write-write conflicts. They then perform revocation, if necessary; acquire the underlying reader-vs-write lock with write permission; execute the writer critical section; and finally release both the mutex and the underlying reader-writer lock. The underlying reader-writer lock resolves read-vs-write conflicts. The code used by readers remains unchanged. This optimization allows readers to make progress during revocation by diverting through the reader slow-path, mitigating the cost of revocation. This also reduces variance for the latency of read operations. We note that this general technique can be readily applied to other existing reader-writer locks that employ distributed reader indicators, such as Linux's brlock [10].

Acknowledgments

We thank Shady Issa for useful discussions about revocation and the cost model. We also thank the anonymous reviewers and our shepherd Yu Hua for providing insightful comments.

References

[1] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management*. ACM, 2011.

[2] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[3] B. B. Brandenburg and J. H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. In *Real-Time Systems Journal*, 2010.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[5] Davidlohr Bueso. futexes and hash table collisions. https://blog.stgolabs.net/2014/01/futexes-and-hash-table-collisions.html, 2014. Accessed: 2019-05-14.

[6] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *Proceedings of ACM PPoPP*, pages 157–166. ACM, 2013.

[7] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, 2008.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2012.

[9] Jonathan Corbet. Driver porting: mutual exclusion with seqlocks. http://lwn.net/Articles/22818, 2003. Accessed: 2018-04-20.

[10] Jonathan Corbet. Big reader locks. https://lwn.net/Articles/378911, 2010. Accessed: 2018-04-20.

[11] Jonathan Corbet. The LRU lock and mmap_sem. https://lwn.net/Articles/753058, 2018. Accessed: 2019-01-10.

[12] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of ACM PPoPP*. ACM, 2012.

[13] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[14] Dave Dice. Biased locking in HotSpot. https://blogs.oracle.com/dave/biased-locking-in-hotspot, 2006.

[15] Dave Dice and Alex Kogan. Compact NUMA-aware locks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[16] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. In *Proceedings of ACM PPoPP*, 2016.

[17] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[18] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[19] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[20] David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Proceedings of the International Conference on Parallel Processing (EuroPar)*. Springer-Verlag, 2013.

[21] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Trans. Parallel Comput.*, 2015.

[22] David Dice, Mark Moir, and William N. Scherer III. Quickly reacquirable locks – US Patent 7,814,488, 2002.

[23] William B. Easton. Process synchronization without long-term interlock. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1971.

[24] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[25] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th*

International Conference on Principles of Distributed Systems, OPODIS'05. Springer-Verlag, 2006.

[26] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings Sixth International Parallel Processing Symposium*, 1992.

[27] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[28] Andi Kleen. Lock elision in the GNU C library. https://lwn.net/Articles/534758, 2013. Accessed: 2019-05-13.

[29] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005.

[30] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[31] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[32] Waiman Long. locking/rwsem: Enable reader optimistic spinning. https://lwn.net/Articles/724384/, 2017. Accessed: 2019-01-24.

[33] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical report, MIT, 2010.

[34] George Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 2003.

[35] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of ACM PPoPP*, 1991.

[36] Oracle. Api documentation for java.util.concurrent.locks.stampedlock. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html, 2012.

[37] Filip Pizlo, Daniel Frampton, and Antony L. Hosking. Fine-grained adaptive biased locking. In *Proceedings of the International Conference on Principles and Practice of Programming in Java (PPPJ)*, 2011.

[38] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2001.

[39] Andreia Craveiro Ramalhete and Pedro Ramalhete. Distributed cache-line counter scalable RW-lock. http://concurrencyfreaks.blogspot.com/2013/09/distributed-cache-line-counter-scalable.html, 2013.

[40] rocksdb.org. A persistent key-value store for fast storage environments. rocksdb.org, 2018.

[41] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[42] Jun Shirako, Nick Vrvilo, Eric G. Mercer, and Vivek Sarkar. Design, verification and applications of a new read-write lock algorithm. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[43] Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 453–472, 2014.

[44] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[45] U. Verner, A. Mendelson, and A. Schuster. Extending Amdahl's law for multicores with turbo boost. *IEEE Computer Architecture Letters*, 2017.

[46] D. Vyukov. Distributed reader-writer mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2011.

[47] Wikipedia contributors. Ski rental problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ski_rental_problem&oldid=813551905, 2017. [Online; accessed 8-August-2018].

[48] Wikipedia contributors. Birthday problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Birthday_problem&oldid=853622452, 2018. [Online; accessed 8-August-2018].

[49] Wikipedia contributors. Loss aversion. https://en.wikipedia.org/wiki/Loss_aversion, 2018.

Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems

Yuanyuan Sun, Yu Hua*, Zhangyu Chen, Yuncheng Guo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

In storage systems, cuckoo hash tables have been widely used to support fast query services. For a read, the cuckoo hashing delivers real-time access with $O(1)$ lookup complexity via open-addressing approach. For a write, most concurrent cuckoo hash tables fail to efficiently address the problem of endless loops during item insertion due to the essential property of hash collisions. The asymmetric feature of cuckoo hashing exhibits fast-read-slow-write performance, which often becomes the bottleneck from single-thread writes. In order to address the problem of asymmetric performance and significantly improve the write/insert efficiency, we propose an optimized Concurrent Cuckoo hashing scheme, called CoCuckoo. To predetermine the occurrence of endless loops, CoCuckoo leverages a directed pseudoforest containing several subgraphs to leverage the cuckoo paths that represent the relationship among items. CoCuckoo exploits the observation that the insertion operations sharing a cuckoo path access the same subgraph, and hence a lock is needed for ensuring the correctness of concurrency control via allowing only one thread to access the shared path at a time; Insertion operations accessing different subgraphs are simultaneously executed without collisions. CoCuckoo improves the throughput performance by a graph-grained locking to support concurrent writes and reads. We have implemented all components of CoCuckoo and extensive experiments using the YCSB benchmark have demonstrated the efficiency and efficacy of our proposed scheme.

1 Introduction

Efficient query services are demanding and important to storage systems, which hold and process much more data than ever and the trend continues at an accelerated pace. The widespread use of mobile devices, such as phones and tablets, accelerates the generation of large amounts of data. There exist 1.49 billion mobile daily active users on Facebook in September 2018, with an increase of 9% year-over-year. In each minute, 300 new profiles are created and more than 208 thousand photos are uploaded to Facebook [5]. The explosion of data volume leads to nontrivial challenge on storage systems, especially on the support for query services [8, 12, 49]. Moreover, write-heavy workloads further exacerbate the storage performance. Much attention has been paid to alleviate the pressure on storage systems, which demands the support of low-latency and high-throughput queries, such as top-*k* query processing [30, 33], optimizing big data queries via automated program reasoning [42], offering practical private queries on public data [47], and optimizing search performance within memory hierarchy [9].

In order to improve the performance of query services for storage systems, efficient hash structures have been widely used. In general, each hash function maps each item to a unique bucket in a hash table, which needs to support constant-scale and real-time access. However, items may be hashed into the same bucket by hash functions, called **hash collisions**. Due to the use of efficient open-addressing design, cuckoo hashing [38] is able to mitigate hash collisions with amortized constant-time insertion overhead and lookup consumption to meet the throughput needs of real-world applications. Unlike conventional hashing schemes that offer only one bucket for each item, the cuckoo hashing provides multiple (usually two in practice [11, 19, 37, 40]) candidate positions for each item to reduce the probability of hash collisions. To perform a lookup operation, at most two positions are probed, and the worst-case time is constant-scale, thus delivering high performance especially in terms of low-latency read and lookup operations [10, 16, 18, 20, 25, 31, 44]. However, to insert an item, the cuckoo hashing has to probe the two candidate buckets for finding an empty position. If an empty slot does not exist, recursive replacement operations are needed to kick items out of their current positions until a vacant bucket is found, which forms a **cuckoo path**. There exists a certain probability of producing an **endless loop**, which occurs after a large number of step-by-step kick-out operations and turn out to be an insertion failure, thus resulting in slow-write performance. The property of asymmetric reads and writes in the cuckoo hashing becomes a potential performance bottlenecks for storage systems.

The endless loops not only lead to the slow-write operation, but also make storage systems unable to efficiently support **concurrent** operations on both reads and writes, which results in poor performance.

As the number of cores is increasing in modern processors, concurrent data structures are promising approaches to provide high performance for storage systems [13, 15, 21, 22, 43, 48, 50]. In general, locks are often utilized to ensure the consistency among multiple threads [4]. To mitigate the hash collisions for writes, most existing hash tables store items in a linked list with coarse-grained locks for entire tables [19], fine-grained locks for per bucket [2, 32] or Read-Copy-Update (RCU) mechanisms [35, 36, 46] for concurrency control. However, the coarse-grained locks for entire tables lead to poor scalability on multi-core CPUs due to the long lock time, and fine-grained per-bucket locks result in substantial resource consumption due to frequent locking and unlocking operations in a cuckoo path. RCU [34] works well for read-heavy workloads, but inefficiently for the workloads with more writes than reads. Moreover, the cuckoo hashing with per-bucket locks suffers from substantial performance degradations due to the occurrence of endless loops. The read operation has to wait for the write operation on the same bucket to complete and then release the lock before being executed. Hence, it is inefficient to frequently lock and unlock buckets during the long cuckoo paths of endless loops.

In order to offer a high-throughput and concurrency-friendly cuckoo hash table, we need to address two main challenges.

Poor Insertion Performance. Cuckoo hashing executes recursive replacing operations to kick items from their storage positions to the candidate positions for finding an empty bucket during an insertion procedure. Moreover, all efforts become useless when encountering an endless loop. To ensure the correctness of concurrency control, two continuous buckets in a path have to be locked for each kick-out operation. The frequent locking and unlocking on a cuckoo path will lead to high time overhead, which decreases the insertion performance.

Poor Scalability. The cuckoo path is possible to be very long in real-world applications. For example, the kick-out threshold is 500 in MemC3 [19], and thus the longest cuckoo path contains 500 buckets. All kick-out operations have to be completed in the buckets protected by the locks. Due to frequent use of locks to ensure consistency among multiple threads, the concurrent hash table results in poor scalability.

In fact, only insertion operations sharing the cuckoo path require locks for ensuring the consistency. Insertion operations through different cuckoo paths can be simultaneously executed. Based on this observation, we propose a lock-efficient concurrent cuckoo hashing scheme, named CoCuckoo. It leverages a directed pseudoforest containing several subgraphs to represent items' hashing relationship, which is further used to indicate the cuckoo paths of insertion operations. In the pseudoforest, each vertex corresponds to a bucket in the hash table, and each edge corresponds to an inserted item from its storage vertex to its backup vertex. In our CoCuckoo, the vertices corresponding to candidate positions of the path-overlapped items must be in the same subgraph. The path-overlapping can be interpreted that two or more cuckoo paths of items share the same nodes during insertion. In our design, each node only exists in a subgraph with a unique subgraph number. Hence, the operations on the path-overlapped nodes access the same subgraph. We leverage a graph-grained, rather than per-bucket, locking scheme to avoid potential contention. Once locking a subgraph number, all buckets with the same number cannot be accessed by different threads at the same time. If one thread intends to access a bucket, it first obtains the number of the subgraph to which the bucket belongs, to check if the subgraph number has been locked.

Specifically, we have the following contributions.

High Throughput. CoCuckoo not only retains cuckoo hashing's strength of supporting constant-scale lookups via open addressing, but also delivers high throughput by a graph-grained locking to support concurrent insertions.

Contention Mitigation. We optimize the graph-grained locking mechanism to pursue low lock overheads for different cases of insertions and release the locks as soon as possible to ease the contention. CoCuckoo is able to predetermine the insertion failures without the need of carrying out continuous kick-out operations, and thus avoids many unnecessary locking operations.

System Implementation. We have implemented all the components and algorithms of CoCuckoo. Moreover, we compared CoCuckoo with state-of-the-art and open-source scheme, *libcuckoo* [32], which offers multi-reader/multi-writer service.

2 Backgrounds

2.1 The Cuckoo Hashing

Cuckoo hashing [38] is an open-addressing technique with $O(1)$ amortized insertion and lookup time. In order to mitigate hash collisions, items can be stored in one of two buckets in a hash table. If one position is occupied, the item can be kicked out to the other [17, 26]. The frequent kick-out operations help items find empty positions for insertion with the costs of possible extra latency. On the other hand, we can definitely read the queried data in one of two hashed positions, thus obtaining constant-scale query time complexity.

Definition 1 ***Conventional Cuckoo Hashing.*** *Suppose that k is the number of hash functions, and S is a set of items. For the case of $k = 2$, conventional cuckoo hashing table H uses two independent and uniformly distributed hash functions h_1, h_2: $S \rightarrow \{0, \ldots, n-1\}$, where n is the size of the hash table. An item x can be stored in any of Buckets $h_1(x)$ and $h_2(x)$ in H, if being inserted successfully.*

The operation for inserting Item x proceeds by computing two hash values of Item x to find Buckets $h_1(x)$ and $h_2(x)$ that could be used to store the item. If either of the two buckets is empty, the item is then inserted into that bucket. Otherwise, an item is randomly chosen from the two candidate buckets and kicked out by Item x. The replaced item is then relocated to its own backup position, possibly replacing another item, until an empty bucket is found or a kick-out threshold is reached. The sequence of replaced items in an insertion operation is called a *cuckoo path*. For example, "$b \to k \to \phi$" is one cuckoo path to identify one bucket available to insert Item x as illustrated in Figure 1, which shows a standard cuckoo hash table with two hash functions. The start point of an edge (arrow) represents the actual storage position of an item, and the end point is the backup position. For instance, the bucket storing Item c is the backup position of Items a and n.

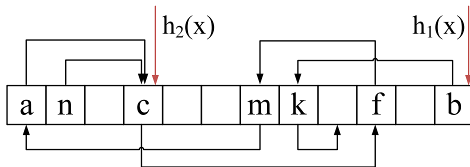


Figure 1: The conventional cuckoo hashing table.

A cuckoo graph is formed by considering each bucket in the hash table as a vertex and each item as an edge. The cuckoo graph can be transformed into a pseudoforest based on graph theory [29].

2.2 Pseudoforest Theory

A pseudoforest is an undirected graph where each vertex only corresponds to at most an edge, and each of maximally connected components, named **subgraphs**, has at most one cycle [7, 23]. The cycle formation starts from a vertex and returns to the vertex through connected edges. Namely, each subgraph in a pseudoforest has no more edges than vertices.

In order to clearly illustrate the direction of a cuckoo path in insertion operations, we extend the pseudoforest into a directed graph by adding the directions from storage positions of items to their backup positions. In the directed pseudoforest, each vertex corresponds to a bucket, and each edge corresponds to an inserted item from the storage vertex to its backup vertex. In the conventional cuckoo hash tables, each bucket stores at most one item, and thus each vertex in a directed pseudoforest has an outdegree of at most one. If the outdegree of a vertex is zero, the vertex corresponds to a vacant bucket, and the subgraph having the vertex contains no cycles, which is called **non-maximal subgraph**. Otherwise, if the outdegrees of all vertices are equal to one, the number of edges of the subgraph is equal to that of vertices, and thus the subgraph contains a cycle, which is a **maximal subgraph**. Any insertion into the subgraph containing a cycle leads to an endless loop [28]. Therefore, if the states of corresponding subgraphs are known before the item is inserted, the insertion result can be predetermined. Based on this property, we can accurately predetermine the occurrence of endless loops without the need of brute-force checking in a step-by-step way.

Figure 2a shows the corresponding pseudoforest of the cuckoo hash table before inserting Item x in Figure 1, which contains one maximal subgraph and one non-maximal subgraph. After inserting Item x, the two subgraphs are merged into a maximal subgraph, as shown in Figure 2b. In particular, after executing the kick-out operations during the insertion, the original vacant vertex becomes the storage position of Item k, and the vertex currently storing Item b becomes the backup position of Item k. The arrow between two vertices needs to be reversed.

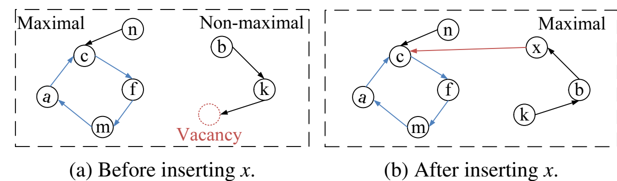


Figure 2: The directed pseudoforest.

In order to maintain the relationship of subgraphs in the pseudoforest, we assign a subgraph number to each bucket and its corresponding vertex, and hence the buckets with the same subgraph number belong to the same subgraph. However, as the subgraphs are merged during item insertion, the vertices with different subgraph numbers are merged into the same subgraph and need to be represented by the same subgraph number. Frequent updates to subgraph numbers of vertices cause severe insertion latency. In order to address this problem, we leverage **the disjoint-set data structure** [24] to maintain the relationship of subgraphs without updating the subgraph numbers.

2.3 The Disjoint-set Data Structure

In general, a disjoint-set data structure [24] is a tree-based structure that handles merging and lookup operations upon disjoint (non-overlapping) subsets, like our design goal. This structure offers near-constant-time complexity in adding new subsets, merging existing subsets, and determining if two or more elements exist in the same subset. Each element in the structure stores an *id*, and a parent pointer. If an element's parent pointer does not point to any other elements, this element is called the root of the tree and becomes the representative member of its subset. A subset may contain only one element. However, if the element has a parent, the element is part of the subset that is identified by uptracking the parents' chain until a representative element (without a parent) is found at the root of the tree.

Three operations can be performed efficiently on the disjoint-set data structure:

MakeSet(x) creates a subset of a new element x, which has a unique *id*, and a parent pointer to itself. The parent

pointer to itself indicates that the element is the representative member of its own subset. The *MakeSet* operation has $O(1)$ time complexity.

Find(x) follows the chain of parent pointers from a leaf element x up to the tree until it reaches the representative member of the tree to which x belongs. In order to make *Find* operations time-efficient, **path compression** operation is used to flatten the tree-based structure by allowing each element to point to the root whenever *Find* is performed. This is valid because each element accessed on the way to the root is part of the same subset. The resulting flatter tree not only speeds up the future operations on these elements, but also accelerates the operations that reference them.

Union(x,y) uses **Find(x)** and **Find(y)** to determine the roots of x and y. If the roots are distinct, the two corresponding trees are combined by attaching the root of one to that of the other, e.g., the root of the tree with fewer elements to the root of the tree having more elements.

The unique number of each subgraph in the directed pseudoforest is viewed as an element in the disjoint-set data structure. When a new subgraph with a unique number is generated, the *MakeSet* operation is called to generate a new corresponding subset. The *Find* operation is performed when we want to know if the subgraphs of two vertices belong to the same subgraph. Moreover, the *Union* operation is triggered when subgraphs are merged.

3 The CoCuckoo Design

The cost-efficient CoCuckoo improves throughput performance via a graph-grained locking to support concurrent insertions and lookups. CoCuckoo leverages a directed pseudoforest containing several subgraphs to represent items' hashing relationship, which is used to indicate the cuckoo paths in insertion operations. A subgraph consists of the vertices corresponding to buckets, as well as the edges corresponding to the inserted items from their storage positions to their backup positions. The disjoint-set data structure is used to maintain the relationship among subgraphs and stored in memory. Meanwhile, the pseudoforest is just a variant of the cuckoo hash table and is not stored. Figure 3 shows the framework of CoCuckoo. A key and its corresponding metadata are stored in each bucket of the hash table (**H_Table**). The metadata per bucket include the position of the value corresponding to the key (*v_pos*), and the subgraph number of the subgraph to which the bucket belongs (*sub_id*). The *sub_id*s are initialized to -1. Moreover, the disjoint-set data structure called **UF_Array** is described in Section 3.3.2.

Each candidate bucket of an item to be inserted into the hash table possibly corresponds to a vertex in the pseudoforest. If a bucket in the hash table does not correspond to any vertices in the pseudoforest, it means that this bucket has not been visited before and is not a candidate bucket for any inserted items. In general, this bucket corresponds to an *EMPTY* subgraph, and its *sub_id* is -1. Hence, an item

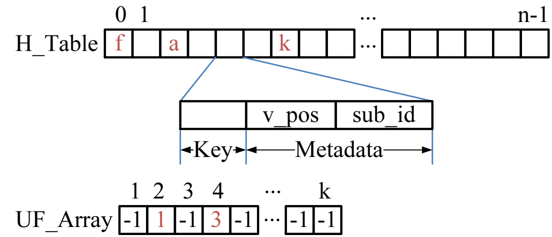


Figure 3: The CoCuckoo framework.

can be directly inserted into a bucket corresponding to an *EMPTY* subgraph. There are at most two *EMPTY* subgraphs for each item insertion due to the existence of two candidate positions. According to the number of corresponding *EMPTY* subgraphs, we classify item insertions into three cases, namely, *TwoEmpty*, *OneEmpty*, and *ZeroEmpty*, respectively showing 2, 1, and 0 *EMPTY* subgraphs.

Moreover, only insertion operations sharing the same cuckoo path require the locks for guaranteeing the correctness of concurrent insertions. Insertion operations through different cuckoo paths access different subgraphs, which can be simultaneously executed and have no lock contention. Based on this observation, CoCuckoo allows the vertices, which correspond to candidate positions of the path-overlapped items, to exist in the same subgraph. We leverage graph-grained locking to avoid potential collisions. These threads won't conflict as long as they manipulate different subgraphs. Most subgraphs are small enough as demonstrated in Figure 6, and hence only a few vertices are constrained at a time.

3.1 Intra-thread Operation

Items inserted into cuckoo hashing form a *cuckoo graph*, which is represented as a directed pseudoforest in our CoCuckoo. Each vertex in the pseudoforest corresponds to a bucket of the hash table and each edge corresponds to an item. An inserted item generates an edge from its actual storage position to its backup position. The pseudoforest reveals cuckoo paths of kick-out operations for item insertion. Hence, the directed pseudoforest can be used to track and exhibit path overlapping of items in advance to avoid potential collisions by a graph-grained locking.

3.1.1 The Case of *TwoEmpty*

When two candidate positions of an item have not yet been used and represented by any vertices in subgraphs of the pseudoforest, i.e., two *EMPTY* subgraphs, the item can be directly inserted into one bucket (the position hashed by the first function by default). Hence, the insertion needs to create a new subgraph, which is non-maximal and ends up with an empty vertex, as shown in Figure 4a to insert Items a, f, and k, respectively. To clearly show these cases, we leverage two hash tables in the following examples. Items hashed by the second function is inserted into the second hash table.

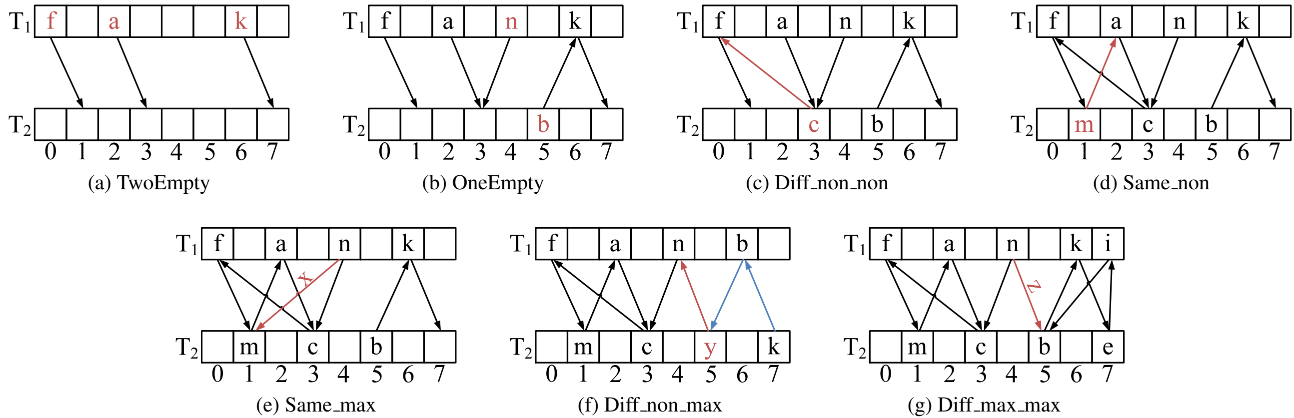


Figure 4: The cases of item insertions.

3.1.2 The Case of *OneEmpty*

One of two candidate buckets of an item corresponds to an existing vertex in the pseudoforest, and the other corresponds to an *EMPTY* subgraph, which will become a new vertex after inserting the item. As shown in Figure 4b, Items b and n can be directly inserted into Buckets $T_2[5]$ and $T_1[4]$ respectively due to available vacancy.

3.1.3 The Case of *ZeroEmpty*

Two candidate positions of an item correspond to two existing vertices in the pseudoforest, which exist either in the same subgraph or both subgraphs. Moreover, each subgraph is either maximal or non-maximal. According to the states and number of subgraphs, we classify the case of *ZeroEmpty* into four subcases.

Diff_non_non: If the vertices corresponding to two candidate positions of an item exist in two non-maximal subgraphs, the item can be successfully inserted into the hash table due to the existence of vacancies. The two subgraphs will be further merged, i.e., inserting Item c into Bucket $T_2[3]$, as shown in Figure 4c.

Same_non: When the two vertices are in the same non-maximal subgraph, there exists a vacant bucket for the item. It will be inserted into the hash table and the corresponding subgraph becomes maximal. For example, Item m is inserted into the hash table and we add an edge from Bucket $T_2[1]$ to Bucket $T_1[2]$. Hence the corresponding subgraph forms a loop, as shown in Figure 4d.

Max: If the two vertices exist in one maximal directed subgraph (named **Same_max**, e.g., Item x in Figure 4e), or two maximal subgraphs (named **Diff_max_max**, e.g., Item z in Figure 4g), no vacancies are available for the new item. The insertion fails even if executing many kick-out operations within a loop. Unlike it, CoCuckoo predetermines the failure in advance and store the item in temporary space (e.g., a stash) [16, 26].

Diff_non_max: One vertex exists in a maximal subgraph and the other is in a non-maximal subgraph in this case. There exists a vacant bucket for the item. The insertion will be successful after several kick-out operations. For example, as shown in Figure 4f, the cuckoo path is "$b \rightarrow k \rightarrow \phi$" when inserting Item y. The two subgraphs are further merged into a new maximal directed subgraph after the insertion.

3.2 Inter-thread Synchronization Optimization

Each thread gets a request from its own task queue each time. Only insertion operations sharing the same cuckoo path require locks for guaranteeing the correctness of concurrent insertions. If two or more insertion requests are path-overlapped, the corresponding threads access to the same subgraph. Hence, the threads that visit the same subgraph have to wait for locks due to guaranteeing the correctness, and the threads that visit different subgraphs can be executed concurrently. To further alleviate the lock contention and overheads, we optimize the operations on different cases of insertions to release locks as soon as possible.

Algorithm 1 shows the steps involved in the insertion of Item x. First, we predetermine the insertion failure if the case is *MAX*, and then lock the subgraphs as shown in Algorithm 2. Second, we determine the case of item insertion and execute corresponding insertion operations.

3.2.1 The Case of *TwoEmpty*

The corresponding thread allocates a new subgraph number, which is locked and assigned to the two *EMPTY* subgraphs. The subgraph number is used to identify a unique subgraph. Since the buckets of *EMPTY* subgraphs can be accessed without acquiring locks, other threads may occupy these buckets before the thread assigns a subgraph number to them. We utilize two atomic subgraph number assignment operations (based on *Compare-And-Swap* instructions) for consistency. If both atomic operations are successful, the two *EMPTY* subgraphs are assigned the same subgraph number. The item can be inserted directly into one of its candidate buckets without kick-out operations. Finally, a non-maximal subgraph is produced. Once the two *EMPTY* subgraphs are

Algorithm 1 Insert(Item x, Hash a, Hash b)

```
1: /*a and b are two indexes of Item x's candidate positions*/
2: if SG_a is maximal && SG_b is maximal then
3:     Return; /*Failure predetermination*/
4: end if
5: LockGraphs(a,b);
6: Result ← False;
7: if SG_a and SG_b are EMPTY then
8:     Result ← InsertTwoEmpty(x,a,b);
9: else if SG_a is EMPTY || SG_b is EMPTY then
10:    if SG_a is EMPTY then
11:        Result ← InsertOneEmpty(x,a,b);
12:    else
13:        Result ← InsertOneEmpty(x,b,a);
14:    end if
15: else
16:    if SG_a is maximal && SG_b is maximal then
17:        Result ← InsertMax(x,a,b);
18:    else if SG_a is non-maximal && SG_b is non-maximal then
19:        if SG_a == SG_b then
20:            Result ← InsertSameNon(x,a,b);
21:        else
22:            Result ← InsertDiffNonNon(x,a,b);
23:        end if
24:    else if SG_a is non-maximal then
25:        Result ← InsertDiffNonMax(x,a,b);
26:    else
27:        Result ← InsertDiffNonMax(x,b,a);
28:    end if
29: end if
30: if Result == False then
31:    Goto 2;
32: end if
```

Algorithm 2 LockGraphs(Int a, Int b)

```
1: while True do
2:     if SG_b < SG_a then
3:         SWAP(SG_a,SG_b);
4:     end if
5:     if SG_a is EMPTY then
6:         Return;
7:     else
8:         /*Lock subgraphs in order to avoid deadlocks*/
9:         LOCK(SG_a);
10:        if SG_a ≠ SG_b then
11:            LOCK(SG_b);
12:        end if
13:    end if
14:    if SG'_a == SG_a&&SG'_b == SG_b then
15:        /*SG'_a and SG'_b are subgraph numbers after locking*/
16:        break;/*Double check*/
17:    else
18:        UNLOCK(SG_a);
19:        if SG_a ≠ SG_b then
20:            UNLOCK(SG_b);
21:        end if
22:    end if
23: end while
```

found with different subgraph numbers, the atomic operations fail, and the *Insert* operation has to be re-executed as shown in Algorithm 3.

Algorithm 3 InsertTwoEmpty(Item x, Hash a, Hash b)

```
1: LOCK(SG); /*The corresponding subgraph number*/
2: if AtomicAssign(&SG_a,SG)&&AtomicAssign(&SG_b,SG) then
3:     DirectInsert(x,Bucket[a]);/*Insert directly into B[a]*/
4:     Tag[SG] ← NON_MAX_MARK;
5:     UNLOCK(SG);
6:     Return True;
7: else
8:     UNLOCK(SG);
9:     Return False;
10: end if
```

3.2.2 The Case of *OneEmpty*

One candidate bucket of the item to be inserted corresponds to an *EMPTY* subgraph, which is vacant. The other candidate bucket of the item corresponds to an existing vertex of a subgraph. The item can be directly inserted into the *EMPTY* subgraph, no matter what the state of another subgraph is. Hence, we utilize two atomic operations without locks to execute the insertion operation. The *Insert* operation atomically assigns the number of the existing subgraph to the new vertex, and inserts the item into the new vertex by an atomic write operation as shown in Algorithm 4. Moreover, the state of the final merged subgraph depends on the pre-merged subgraph without changes. If the vertex has been already occupied by another item, the subsequent atomic write operation of insertion fails, which means that the original *EMPTY* subgraph has been merged with another subgraph and becomes not empty. The *Insert* operation has to be restarted, and the insertion case becomes *ZeroEmpty*. Hence, the insertion protocol ensures forward progress and doesn't produce repeated atomic operation failures.

Algorithm 4 InsertOneEmpty(Item x, Hash a, Hash b)

```
1: if AtomicAssign(&SG_a,SG_b) then
2:     if AtomicInsert(x,Bucket[a]) then
3:         Return True;
4:     else
5:         Return False;
6:     end if
7: else
8:     Return False;
9: end if
```

3.2.3 The Case of *ZeroEmpty*

Diff_non_non: The *Insert* operation locks the two corresponding non-maximal subgraphs, inserts the item after several kick-out operations, and then releases the lock after the

two subgraphs have been merged as shown in Figure 5. The merged subgraph is non-maximal, which has a vacant bucket for another item to be inserted.

Algorithm 5 InsertDiffNonNon(Item x, Hash a, Hash b)

1: $Kick\text{-}out(x, Bucket[a])$; /*Enter from $B[a]$*/
2: $Union(SG_a, SG_b)$;
3: $UNLOCK(SG_a)$;
4: $UNLOCK(SG_b)$;
5: Return $True$;

Same_non: The merged subgraph is maximal after the item is inserted, and no vacancies are available for other items to be inserted. Hence, to ease the lock contention, our optimization is to lock the corresponding subgraph and mark it to be maximal, and unlock the subgraph before executing kick-out operations of the insertion, as shown in Algorithm 6.

Algorithm 6 InsertSameNon(Item x, Hash a, Hash b)

1: $Tag[SG] \leftarrow MAX_MARK$;
2: $UNLOCK(SG)$;
3: $Kick\text{-}out(x, Bucket[a])$;/*Enter from $B[a]$*/
4: Return $True$;

Max: No vacancies are available in the corresponding subgraphs for items in this case, and the *Insert* operation will always walk into a loop and be predetermined to a failure. We just unlock the corresponding subgraph(s) without any other operations, as shown in Algorithm 7.

Algorithm 7 InsertMax(Item x, Hash a, Hash b)

1: $UNLOCK(SG_a)$;
2: **if** $SG_a \neq SG_b$ **then**
3: $\quad UNLOCK(SG_b)$;
4: **end if**
5: Return $True$;

Diff_non_max: There exists only one vacancy for an item, and the state of the merged subgraph is predetermined to be maximal after the item is inserted in the case. Other threads accessing the subgraph first obtain the subgraph state and will not insert items when the subgraph is maximal. For the *Insert* operation, the corresponding thread obtains the lock of the non-maximal subgraph and marks it to be maximal, and then releases the lock immediately. The insertion with several kick-out operations and the merging operation of two subgraphs complete outside the lock, as shown in Algorithm 8.

Algorithm 8 InsertDiffNonMax(Item x, Hash a, Hash b)

1: $Tag[SG_a] \leftarrow MAX_MARK$;
2: $UNLOCK(SG_a)$;
3: $UNLOCK(SG_b)$;
4: $Kick\text{-}out(x, Bucket[a])$;/*Enter from $B[a]$*/
5: $Union(SG_a, SG_b)$;
6: Return $True$;

3.3 Subgraph Management

3.3.1 Subgraph Number Allocation

We allocate each newly created subgraph a new number for identification. Each subgraph number represents a unique subgraph, and each vertex of the subgraph records the number of its corresponding bucket. All threads allocate subgraph numbers concurrently. When a thread requests a subgraph number, we lock the number generator. Other threads have to wait for unlocking, thus increasing the response time of requests and possibly becoming performance bottleneck. Moreover, we only need to confirm that the subgraph numbers are unique without the need of continuity. In order to decrease the response time without locks and ensure consistency of subgraph number allocation, we leverage a simple modular function to compute the subgraph numbers for all threads. In the modular function, the modulus is the total number of threads p, and the remainder is the number of each thread r. Hence, the subgraph number allocated by each thread is $n = kp + r$, while the parameter k is an accumulator. A subgraph number generator serves for a thread. For example, in the 8-thread CoCuckoo, the subgraph numbers allocated by $Thread$ 2 is 2, 10, 18, and so on. Hence, we allocate subgraph numbers for each thread in order, and then add one to the accumulator k after creating a new subgraph.

3.3.2 Subgraph Merging

When vertices corresponding to two candidate positions of an inserted item exist in two subgraphs, they are merged after item insertion. To avoid exhaustively searching for all vertex members of corresponding subgraphs and updating their subgraph numbers (sub_ids in metadata) when subgraphs are merged, a tree-based data structure called **disjoint-set data structure** is utilized to maintain the relationship between subgraphs. Each node in the tree stores a subgraph number and a parent pointer. In order to avoid deadlocks, we always merge the subgraphs with bigger numbers into that with smaller numbers. After subgraphs are merged, the parent pointer of the node with bigger number points to the node with smaller number, which becomes the representative of the tree. The newly merged subgraph is possible to be merged again with others. Finally, the subgraph number of the tree's root represents the number of the subgraph merged from all prior subgraphs.

The $Union$ operation is called when merging subgraphs is needed. $Union(sub_id1, sub_id2)$ uses $Find(sub_id1)$ and $Find(sub_id2)$ to determine the roots of the nodes with sub_id1 and sub_id2. If the roots are distinct, the trees are combined by attaching the tree whose root has the bigger subgraph number to the root of the smaller one. Furthermore,

$Find(sub_id)$ follows the chain of parent pointers from a leaf node with sub_id1 up to the tree until it reaches a root node, whose parent is itself. The subgraph number stored in the root node is the actual subgraph number of all nodes in the chain.

An array **UF_Array** is used to implement the disjoint-set data structure. The length of the array is the number of subgraphs, and the indexes of the array indicate the subgraph numbers. The values of elements are initialized to -1, and updated to the indexes of their parents after merging. Specifically, there are two cases for the value of an element in the array: (1) If the value of an element is equal to -1, the subgraph number represented by the index is a root node; (2) If the value of an element is larger than 0, the subgraph number corresponding to the value is the parent node of the subgraph number represented by the index. As shown in Figure 3, $UF_Array[1] = -1$ means that the subgraph number 1 is the root node; $UF_Array[2] = 1$ means that the subgraph number 1 is the parent node of the subgraph number 2. Therefore, all buckets with subgraph number 2 have the actual subgraph number of 1.

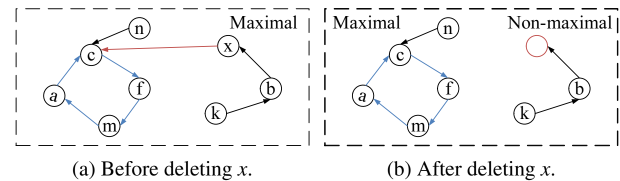
3.3.3 Item Deletion and Subgraph Splitting

For an item, when the vertex of the storage position is not on a cycle, the corresponding subgraph is split into two subgraphs when deleting the item [6]. To avoid reconstructing the hash table and updating *sub_ids* of all buckets, the *re_edge* information in metadata per bucket is added to record the related edges of the bucket in the corresponding subgraph. When deleting Item *x*, we first compute the two corresponding buckets of Item *x*, i.e., determine the storage and backup positions. Item *x* is deleted from the storage bucket in the hash table, and the corresponding edge is deleted from the *re_edge* in two buckets. All related edges and buckets are then recursively searched from the *re_edge* in the candidate buckets. Finally, all searched buckets are updated with a new *sub_id*. In order to optimize the delete operation after splitting subgraphs, all buckets in the subgraph containing the storage position of Item *x* do not need to update their *sub_ids*. Their actual *sub_ids* can be searched by UF_Array. The thread that performed the delete operation acquires the lock of the corresponding subgraph until the operation completes. Figure 5 shows an example of splitting subgraph when deleting Item *x*. The backup position of Item *x* (namely, the storage position of Item *c*) is found by hash computation, and all related nodes in the left subgraph are iteratively searched by the *re_edge* information in the metadata of related buckets. We then update all *sub_id*s of searched related buckets in hash tables.

4 Performance Evaluation

4.1 Experimental Setup

The server used in our experiments is equipped with an Intel 2.4GHz 16-core CPU, 24GB DDR3 RAM, and 2TB hard disk. The L1 and L2 caches of the CPU are 32KB and 256KB, respectively. The programming language of all functional components of CoCuckoo scheme is C/C++. Moreover, multi-threading is implemented by the *pthread* via a *pthread.h* header and a thread library.



(a) Before deleting *x*. (b) After deleting *x*.

Figure 5: An example of splitting subgraph.

Workloads: The widely-used industry standard in evaluating the performance of key-value stores is the *Yahoo! Cloud Serving Benchmark* (YCSB) [14]. We use this benchmark to generate five workloads, each with different proportions of *Insert* and *Lookup* queries to represent real-world scenarios, as shown in Table 1. Especially, the INS workload is the worst case for cuckoo hashing (resulting in a nearly full table), and the remaining four workloads are common cases with *Lookup* operations. Moreover, each workload has two million key-value pairs. Each key in workloads is 16 bytes and each value is 32 bytes for most experiments (except in Section 4.2.4 that uses various sizes for evaluating the effect of size on throughput). The default cuckoo hash table has $2^{21} = 2,097,152$ slots, which consumes about 96MB memory in total.

Table 1: Distributions of different queries in each workload.

Workload	Insert	Lookup
Insert-only (INS)	100%	0%
Insert-heavy (IH)	75%	25%
Insert-lookup balance (ILB)	50%	50%
Lookup-heavy (LH)	25%	75%
Lookup-only (LO)	0%	100%

Threads: In our experiments, there are five settings for the number of threads, including a single thread, 4, 8, 12, and 16 threads, to evaluate the concurrency performance.

Comparisons: We compare our proposed CoCuckoo with open-source *libcuckoo* [3, 32], which is optimized to offer multi-reader/multi-writer service through spin_locks based on concurrent multi-reader/single-writer cuckoo hashing used in *MemC3* [19]. Since *libcuckoo* has multiple slots per bucket to mitigate collisions [19,39,41,51], we also evaluate the performance of 2-, 4-, 8- and 16-way *libcuckoo*. We follow the default configuration of *libcuckoo* in the original paper [32] and only adjust the numbers of threads and slots to facilitate fair comparisons in concurrency. The results of CoCuckoo and *libcuckoo* come from in the same experimental environment.

We focus on the performance improvements from our design in the context of workloads with concurrent insertions and lookups by measuring the throughput and latency of

multiple threads accessing the same hash table. In general, the request response on the cuckoo hashing table becomes slower as the load factor increases, since more items have to be moved [32]. Hence, we measure the throughput and latency for certain load factor intervals (from 0 to 80%), and average throughput and latency.

4.2 Results and Analysis

4.2.1 Lock Granularity

A graph-grained lock is used for concurrency control in our CoCuckoo. Since most subgraphs are small, the granularity of graph-grained locks is acceptable, which only constrain a very small number of buckets. We have measured the number of subgraphs in each size interval with Insert-only workload. Figure 6 demonstrates that most subgraphs are small. For example, 44.25% subgraphs contain only 3 vertices, and about 99% subgraphs contain no more than 10 vertices. Very few buckets are constrained once for ensuring the correctness of concurrency control of the multi-thread hash tables. We obtained identical experimental results under single-threaded and multi-thread conditions. The reason is that for given hash functions, the hash location of each item is determined. Based on hashed locations, the cuckoo subgraphs are also determined, except the difference of the order in which the vertices are added into subgraphs due to the concurrency of threads.

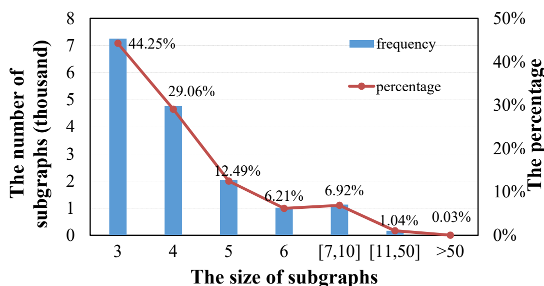


Figure 6: The number of subgraphs in each size interval.

4.2.2 Throughput

This subsection evaluates the average throughput under an increasing number of threads with five workloads and throughput at different table occupancies with Insert-only workload. The throughput is the average number of requests completed per second.

Figure 7 shows the average throughput under an increasing number of threads with five different workloads. The average throughput increases with the increasing number of threads in all cuckoo hashing tables, due to the multi-thread acceleration. In *libcuckoo*, the lower associativity improves the throughput, because each lookup checks fewer slots in order to find the key, and each insertion needs to probe fewer slots in a bucket for an empty slot. In 4-way *libcuckoo*, each *Lookup* requires at most two cache-line reads to find the key and one more cache-line read to obtain the value. Furthermore, in 16-way *libcuckoo*, each *Lookup* requires at most eight cache-line reads to find the key and one more cache-line read to obtain the value. With the increasing number of threads, we observe that CoCuckoo significantly increases the average throughput over *libcuckoo* by 50% to 130% as shown in Figure 7f. In particular, the growth rate is defined as the throughput growth ratio of CoCuckoo relative to 2-way *libcuckoo*.

Figure 8a illustrates that the impact of load factors on 16-thread cuckoo hashing throughput for Insert-only workload. With the increase of load factors, the throughput decreases, since each *Insert* operation has to probe more buckets for finding an empty slot, and requires more item replacements to insert the new item. The *libcuckoo* utilizes *Breadth-First Search* (BFS) to find an empty slot for item insertion, and the path threshold is 5 in open-source implementation [3]. In 1-way *libcuckoo*, the threshold is reached at lower load factor, and expensive rehashing operations are executed, which results in poor performance. However, a higher load factor (more than 0.65) results in performance decrease in terms of throughputs of CoCuckoo. We argue that as the load factor increases, the subgraph merge operations become more frequent, thus leading to the decreasing number of subgraphs and the increasing number of vertices included in a single subgraph. Hence, more vertices are constrained by one thread that handles the *Insert* operation due to the graph-grained locking. Compared with *libcuckoo*, the proposed CoCuckoo obtains significant performance improvements in terms of throughputs.

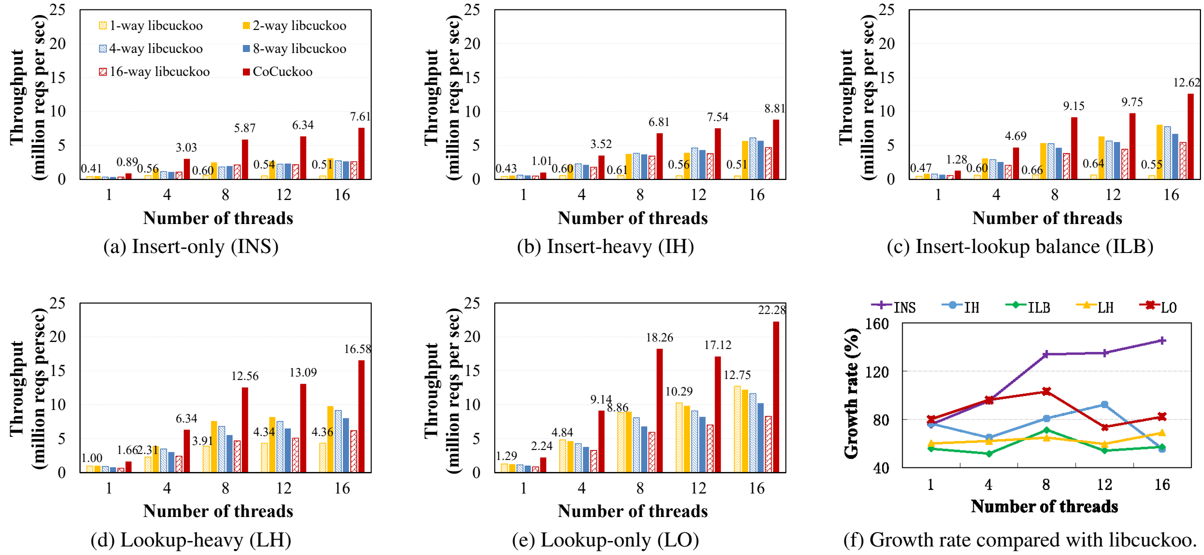
4.2.3 Predetermination for Insertion

Table 2 shows the fractions of all cases with different workloads of containing *Insert* operations in 16 threads. In each workload, the two cases of *TwoEmpty* and *OneEmpty* account for a large proportion, which means that most insertion operations probe new and empty buckets, and add new vertices into corresponding subgraphs in the pseudoforest. Hence, with executing these insertion operations, the threads leverage short-term (*TwoEmpty*) or no (*OneEmpty*) locks the shared buckets, which alleviates the lock contention and further improves the throughput of CoCuckoo. The *Max* case occurs in INS and IH workloads, which means the proposed CoCuckoo predetermines insertion failures and releases locks without any kick-out operations to ease the contention.

4.2.4 Different Key Sizes

All prior experiments used the workloads with 16-byte keys and 32-byte values. We further evaluate the average throughput of cuckoo hash tables with different key sizes at the load factor of 0.8, as shown in Figure 8b- 8f.

In each figure, we show the average throughput, as the key size increases from 8 bytes to 64 bytes with Insert-only workload. The throughput decreases, as the key size increases due to the increased *String_copy* and *String_comp* overheads, as well as memory bandwidth overhead. More-



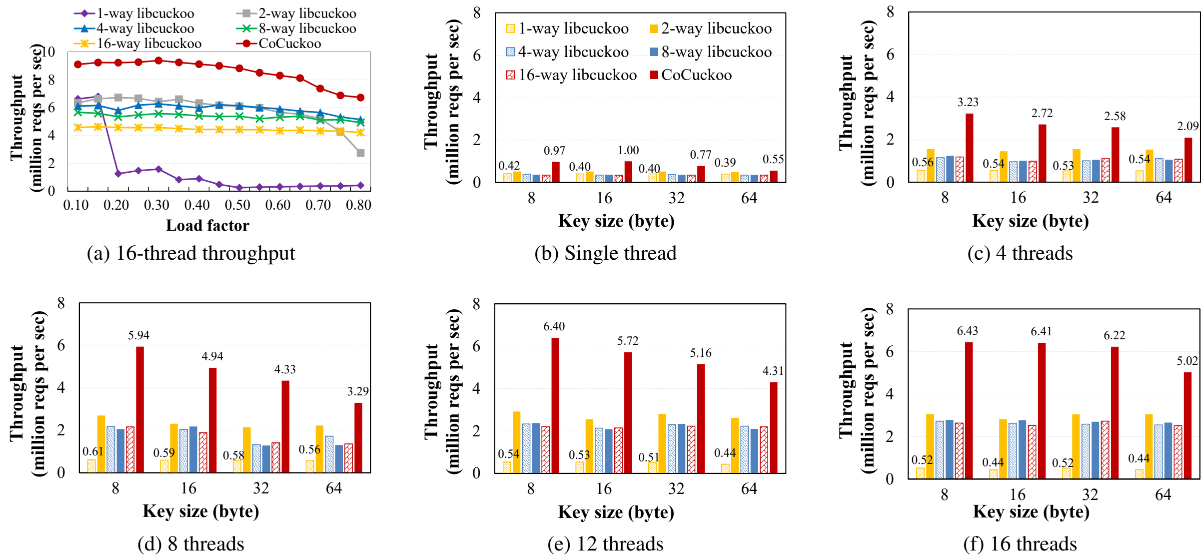
(a) Insert-only (INS) (b) Insert-heavy (IH) (c) Insert-lookup balance (ILB)

(d) Lookup-heavy (LH) (e) Lookup-only (LO) (f) Growth rate compared with libcuckoo.

Figure 7: The average throughput and growth rate compared with 2-way *libcuckoo*.

Table 2: The fractions of all cases in 16 threads.

Workloads	TwoEmpty	OneEmpty	Same_non	Max	Diff_non_non	Diff_non_max
Insert-only	25.673%	37.9628%	0.0003%	13.9802%	13.1447%	9.239%
Insert-heavy	32.9343%	40.4907%	0.0004%	3.5921%	16.7513%	6.2312%
Insert-lookup balance	44.675%	39.6011%	0.0002%	0%	15.7235%	0.0002%
Lookup-heavy	64.4448%	30.1658%	0%	0%	5.3894%	0%



(a) 16-thread throughput (b) Single thread (c) 4 threads

(d) 8 threads (e) 12 threads (f) 16 threads

Figure 8: The throughput in different table occupancies and sizes of keys with Insert-only workload.

over, our concurrent cuckoo hashing becomes much less effective with large keys. For example, in the case of 16 threads, the throughput of CoCuckoo is 6.43 million requests per second with 8-byte keys, which is 135% higher than 4-way *libcuckoo*. The throughput of CoCuckoo is only 95% higher than 4-way *libcuckoo* with 64-byte keys. Similarly, hyperthreading also becomes much less effective with larger keys. For example, with 64-byte keys, the 4-thread throughput of CoCuckoo is 2.09 million requests per second, 8-thread throughput is more than 57% higher than 4-thread throughput, but 16-thread throughput is only 50% higher than 8-thread throughput. In order to support large-size keys, the full keys and values can be stored outside the table and referenced by a pointer, which possibly damages lookup performance. A null pointer indicates that the bucket is unoccupied.

4.2.5 Extra Space Overhead and Impact

Extra space overhead comes from the auxiliary structure of CoCuckoo, which includes two parts. One is the UF_Array to maintain the relationship among subgraphs. The other is the *sub_id* (subgraph number) stored in each bucket of the cuckoo hash table. Specifically, the UF_Array length is the number of subgraphs, and the indexes of the array indicate the subgraph numbers. The maximum number of subgraphs is equal to that of buckets in hash tables. The subgraph number is an *Int* type data in our implementation, which is usually 4 bytes in currently compilers (e.g., GCC). For subgraph numbers, the *sub_id* is stored in each bucket as metadata.

The default cuckoo hash table has 2^{21} slots. Therefore, the extra space overhead is totally $2^{21} * (4+4)B = 16MB$, which is deterministic and very small, compared with current memory capacity. Moreover, the pseudoforest is the theoretical transformation form of the cuckoo hash table, which is not stored in memory. In essence, we leverage acceptable space overhead to obtain significant performance improvements, which is a suitable trade-off. Moreover, in order to examine its impact upon system performance, we evaluate the throughput with the extra space through the Insert-only workload. To facilitate fair comparisons, we define the identical space available for both *libcuckoo* and CoCuckoo. As shown in Figure 9, we observe that CoCuckoo increases the throughput over 2-way *libcuckoo* by 73% to 159%, which is comparable to 75%-150% in Figure 7a in Section 4.2.2, thus exhibiting little impact.

We also evaluate the average execution time per request. As shown in Figure 10, the average time per request in 16-thread CoCuckoo is $1.66\mu s$, which is much shorter than $16.37\mu s$ in *libcuckoo*. Frequent rehash operations occur during the item insertion of 1-way *libcuckoo*, resulting in longer insertion time.

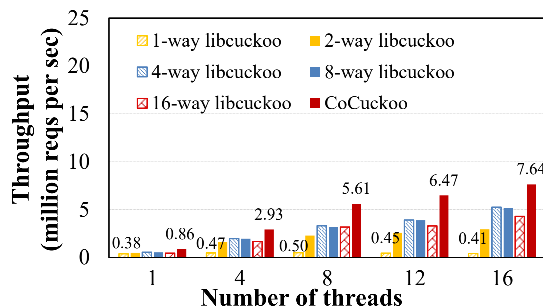


Figure 9: The average throughput with the same space overhead at the load factor of 0.8.

4.2.6 Deletion Latency

The *re_edge* information in metadata per bucket is included for supporting deletion. We evaluate the impact of deletion on performance by measuring request latency. The latency is defined as the time required to be executed in the concurrent program operation per request except the time in a serial program operation. Specifically, for a request, the latency is equal to the execution time of a thread in a concurrent program minus the time it consumes in a serial program. We use the YCSB benchmark to generate two workloads: (1) 5% *Delete*: 95% *Insert* and 5% *Delete* requests; (2) 10% *Delete*: 90% *Insert* and 10% *Delete* requests, each with one million key-value pairs with deletion.

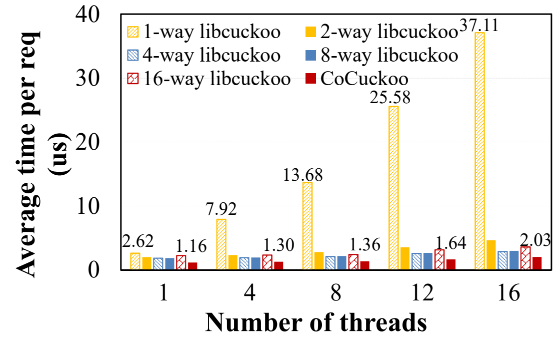


Figure 10: The average time per request with the same space overhead.

Figure 11 shows the average request latency of CoCuckoo with six workloads containing *Insert* and *Delete* operations. With the increasing number of threads, the average latency increases due to more intense lock contentions. The time waiting for locks increases with the increasing number of threads. For example, with Insert-only workload, the single-thread latency is $1.16\mu s$ per request, 8-thread latency is 2.6% longer than single-thread latency, and 16-thread latency is 50.0% longer than 8-thread latency. However, with the same number of threads, the average latency of different workloads incurs slight changes. Deletions are generally executed within locks, which exacerbates lock contentions. The latency of *Delete* is similar to that of *Insert* and larger than that of *Lookup*.

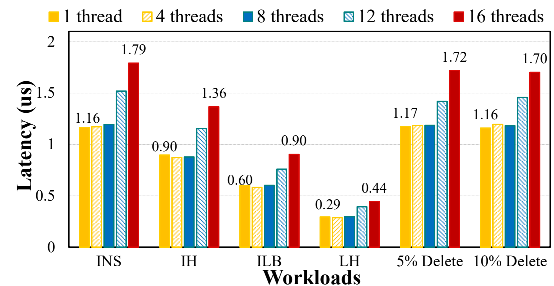


Figure 11: Average latency in different workloads.

Figure 12 illustrates the impact of load factors on latency performance of CoCuckoo with 5% *Delete*. As the load factor increases, there is a slight decrease in latency. Due to our optimization in Section 3.2.2, the average latency of Case *OneEmpty* is smaller than that of Case *TwoEmpty*. The proportion of Case *OneEmpty* increases while the proportion of Case *TwoEmpty* significantly decreases as the load factor increases. Hence, the overall latency decreases. In the meantime, due to appropriate load factors, the decrease of request latency is slow. For example, 16-thread latency is $1.73\mu s$ at

a load factor of 0.10 for the hash table, and 1.68μs at a load factor of nearly 0.50, which is only 2.9% smaller than that at a load factor of 0.10.

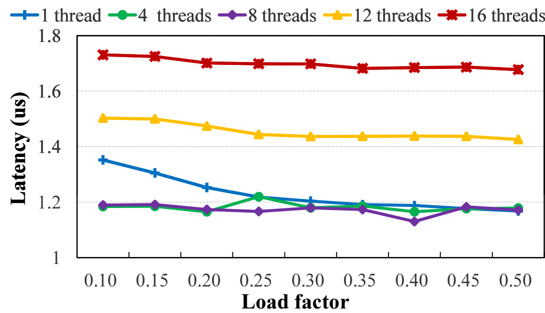


Figure 12: The latency at different table occupancies with 5% *Delete*.

5 Related Work

Performance-constrained single-thread hash tables. The Google's *dense_hash_map*, which is available in the Google SparseHash [1] library, supports fast lookup services. The dense hash uses open addressing with internal quadratic probing and achieves space efficiency for extremely high speeds. Moreover, this table stores items in a single large array and maintains a maximum 0.5 load factor by default.

Horton table [11] is an enhanced bucketized cuckoo hash table for reducing the number of CPU cache lines that are accessed in each lookup, thus achieving higher throughput. Most items are hashed by only a single function and therefore are retrieved by accessing a single bucket, namely, a single cache line. For negative lookups, item remapping enables low access costs to access one cache line.

SmartCuckoo [45] is a cost-efficient cuckoo hash table for accurately predetermining the status of cuckoo operations and the occurrence of endless loops. A directed pseudoforest representing the hashing relationship is utilized to track item placements in the hash table, and further avoid walking into an endless loop.

However, the performance of these hash tables does not scale with the number of cores in the processor, due to only a single thread permitted in execution, which suffer from the performance bottleneck of slow writes in storage systems. Unlike them, the design goal of our CoCuckoo is to efficiently support concurrent operations and deliver high performance in storage systems via concurrency-friendly hash tables.

High-performance concurrent hash tables. *Relativistic hash table* [46] is the data structure that supports shrinking and expanding while allowing concurrent, wait-free and linearly scalable lookups. The proposed resize algorithms reclaim memory as the number of items decreases, and enable Read-Copy Update (RCU) [34–36, 46] hash tables to maintain constant-scale performance as the number of item increases, without delaying or disrupting readers.

MemC3 [19] is designed to provide caching for read-mostly workloads and leverages the optimistic cuckoo hashing, which supports multiple readers without locks and a single writer. Instead of moving "items" forward along the cuckoo path, the cuckoo hashing used in *MemC3* moves "vacancies" backwards along the cuckoo path. The backward method ensures that an item can always be found by a reader.

The *libcuckoo* [32] redesigns *MemC3* to minimize the size of the hash table's critical sections and allow for significantly increased parallelism, which supports two concurrency control mechanisms, i.e., fine-grained locking and hardware transactional memory.

These schemes are dedicated to improve lookup performance and however fail to work well for tables with more insertion than lookup operations due to the occurrence of hash collisions and endless loops. Our proposed CoCuckoo focuses on write-heavy workloads for supporting concurrent insertions and lookups.

Moreover, CoCuckoo currently addresses the performance bottleneck for cuckoo hashing with two hash functions. The setting of more than two hash functions would significantly increase operation complexity [16,52], which can be reduced to two using techniques such as double hashing [27].

6 Conclusion

Most existing concurrent cuckoo hash tables are used for read-intensive workloads and fail to address the potential problem of endless loops during item insertion. We proposed CoCuckoo, an optimized concurrent cuckoo hashing scheme, which represents cuckoo paths as a directed pseudoforest containing multiple subgraphs to indicate items' hashing relationship. Insertion operations sharing the same cuckoo path need to access the same subgraph, and hence a lock is needed for ensuring the correctness of concurrent operations. Insertion operations accessing different subgraphs enable simultaneous execution. CoCuckoo classifies item insertions into three cases and leverages a graph-grained locking mechanism to support concurrent insertions and lookups. We further optimize the mechanism and release locks as soon as possible to mitigate the contention for pursuing low lock overheads. Extensive experiments using the YCSB benchmark demonstrate that the proposed CoCuckoo achieves higher throughput performance than state-of-the-work scheme, i.e., *libcuckoo*.

7 Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, and National Natural Science Foundation of China (NSFC) under Grant No. 61772212. The authors are grateful to anonymous reviewers and our shepherd, Haris Volos, for their constructive feedbacks and suggestions.

References

[1] Google SparseHash. https://github.com/sparsehash/sparsehash.

[2] Intel Threading Building Block. https://www.threadingbuildingblocks.org/.

[3] Libcuckoo library. https://github.com/efficient/libcuckoo.

[4] Memcached. A distributed memory object caching system. http://memcached.org/, 2011.

[5] The Top 20 Valuable Facebook Statistics. https://zephoria.com/top-15-valuable-facebook-statistics/, Updated March 2019.

[6] ALSTRUP, S., GØRTZ, I. L., RAUHE, T., THORUP, M., AND ZWICK, U. Union-Find with Constant Time Deletions. In *International Colloquium on Automata, Languages, and Programming* (2005), Springer, pp. 78–89.

[7] ÀLVAREZ, C., BLESA, M., AND SERNA, M. Universal Stability of Undirected Graphs in the Adversarial Queueing Model. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)* (2002), ACM, pp. 183–197.

[8] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A View of Cloud Computing. *Communications of the ACM 53*, 4 (April 2010), 50–58.

[9] AYERS, G., AHN, J. H., KOZYRAKIS, C., AND RANGANATHAN, P. Memory Hierarchy for Web Search. In *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), IEEE, pp. 643–656.

[10] BRESLOW, A. D., AND JAYASENA, N. S. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment 11*, 9 (2018), 1041–1055.

[11] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., AND TULLSEN, D. M. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)* (2016), USENIX Association, pp. 281–294.

[12] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)* (2011), ACM.

[13] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box Concurrent Data Structures for NUMA Architectures. In *Proc. ASPLOS* (2017), ACM, pp. 207–221.

[14] 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 (SoCC)* (2010), ACM, pp. 143–154.

[15] DAVID, T., DRAGOJEVIC, A., GUERRAOUI, R., AND ZABLOTCHI, I. Log-free concurrent data structures. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC)* (2018), USENIX Association, pp. 373–386.

[16] DEBNATH, B. K., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of 2010 USENIX Annual Technical Conference (USENIX ATC)* (2010), pp. 1–16.

[17] DEVROYE, L., AND MORIN, P. Cuckoo hashing: Further analysis. *Information Processing Letters 86*, 4 (2003), 215–219.

[18] EPPSTEIN, D., GOODRICH, M. T., MITZENMACHER, M., AND TORRES, M. R. 2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)* (2017), ACM, pp. 247–260.

[19] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the Symposium on Network System Design and Implementation (NSDI)* (2013), vol. 13, pp. 385–398.

[20] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), ACM, pp. 75–88.

[21] FATOUROU, P., KALLIMANIS, N. D., AND ROPARS, T. An Efficient Wait-free Resizable Hash Table. In *Proc. SPAA* (2018), ACM.

[22] FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PETRANK, E. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proc. PPoPP* (2018), ACM, pp. 28–40.

[23] GABOW, H. N., AND WESTERMANN, H. H. Forests, Frames, and Games: Algorithms for Matroid Sums and Applications. *Algorithmica 7*, 1 (1992), 465–497.

[24] GALLER, B. A., AND FISHER, M. J. An Improved Equivalence Algorithm. *Communications of the ACM 7*, 5 (1964), 301–303.

[25] HUA, Y., XIAO, B., AND LIU, X. Nest: Locality-aware Approximate Query Service for Cloud Computing. In *Proc. INFOCOM* (2013), IEEE, pp. 1303–1311.

[26] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[27] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Pearson Education, 1997.

[28] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient Parallel Algorithms for Graph Problems. *Algorithmica 5*, 1 (1990), 43–64.

[29] KUTZELNIGG, R. Bipartite Random Graphs and Cuckoo Hashing. In *Discrete Mathematics and Theoretical Computer Science* (2006), Discrete Mathematics and Theoretical Computer Science, pp. 403–406.

[30] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (2009), vol. 9, pp. 153–166.

[31] LI, Q., HUA, Y., HE, W., FENG, D., NIE, Z., AND SUN, Y. Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services. In *Proceedings of the 22nd International Symposium of Quality of Service (IWQoS)* (2014), IEEE, pp. 153–158.

[32] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), ACM, p. 27.

[33] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (2009), vol. 9, pp. 111–123.

[34] MCKENNEY, P. E. RCU vs. Locking Performance on Different CPUs. In *linux.conf.au* (2004).

[35] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-Copy Update. In *Ottawa Linux Symposium* (2002), pp. 338–367.

[36] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[37] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[38] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[39] POLYCHRONIOU, O., RAGHAVAN, A., AND ROSS, K. A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. SIGMOD* (2015), ACM, pp. 1493–1508.

[40] RICHA, A. W., MITZENMACHER, M., AND SITARAMAN, R. The Power of Two Random Choices: A Survey of Techniques and Results. *Combinatorial Optimization 9* (2001), 255–304.

[41] ROSS, K. A. Efficient Hash Probes on Modern Processors. In *IEEE 23rd International Conference on Data Engineering (ICDE)* (2007), IEEE, pp. 1297–1301.

[42] SCHLAIPFER, M., RAJAN, K., LAL, A., AND SAMAK, M. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017), ACM, pp. 631–646.

[43] SHAVIT, N. Data Structures in the Multicore Age. *Communications of the ACM 54*, 3 (2011), 76–84.

[44] SUN, Y., HUA, Y., FENG, D., YANG, L., ZUO, P., AND CAO, S. MinCounter: An Efficient Cuckoo Hashing Scheme for Cloud Storage Systems. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–7.

[45] SUN, Y., HUA, Y., JIANG, S., LI, Q., CAO, S., AND ZUO, P. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC)* (2017), USENIX Association, pp. 553–565.

[46] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC)* (2011), USENIX, pp. 145–158.

[47] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical Private Queries on Public Data. In *Proc. NSDI* (2017), pp. 299–313.

[48] WINBLAD, K., SAGONAS, K., AND JONSSON, B. Lock-free Contention Adapting Search Trees. In *Proc. SPAA* (2018), ACM, pp. 121–132.

[49] WU, S., LI, F., MEHROTRA, S., AND OOI, B. C. Query Optimization for Massively Parallel Data Processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)* (2011), ACM.

[50] WU, Y., AND TAN, K.-L. Scalable In-Memory Transaction Processing with HTM. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)* (2016), pp. 365–377.

[51] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[52] ZUO, P., AND HUA, Y. A Write-friendly Hashing Scheme for Non-volatile Memory Systems. In *Proc. MSST* (2017).

NICA: An Infrastructure for Inline Acceleration of Network Applications

Haggai Eran[1,2], Lior Zeno[1], Maroun Tork[1], Gabi Malka[1], and Mark Silberstein[1]

[1]*Technion – Israel Institute of Technology* [2]*Mellanox Technologies*

Abstract

With rising network rates, cloud vendors increasingly deploy FPGA-based SmartNICs (F-NICs), leveraging their *inline* processing capabilities to offload hypervisor networking infrastructure. However, the use of F-NICs for *accelerating general-purpose server applications in clouds* has been limited.

NICA is a hardware-software co-designed framework for inline acceleration of the application data plane on F-NICs in multi-tenant systems. A new *ikernel* programming abstraction, tightly integrated with the network stack, enables application control of F-NIC computations that process *application* network traffic, with minimal code changes. In addition, NICA's virtualization architecture supports fine-grain time-sharing of F-NIC logic and provides I/O path virtualization. Together these features enable cost-effective sharing of F-NICs across virtual machines with strict performance guarantees.

We prototype NICA on Mellanox F-NICs and integrate ikernels with the high-performance VMA network stack and the KVM hypervisor. We demonstrate significant acceleration of real-world applications in both bare-metal and virtualized environments, while requiring only minor code modifications to accelerate them on F-NICs. For example, a transparent key-value store cache ikernel added to the stock `memcached` server reaches 40 Gbps server throughput (99% line-rate) at 6 µs 99th-percentile latency for 16-byte key-value pairs, which is 21× the throughput of a 6-core CPU with a kernel-bypass network stack. The throughput scales linearly for up to 6 VMs running independent instances of `memcached`.

1 Introduction

SmartNICs with integrated FPGAs (*F-NICs*) [19, 68, 89, 119] are an appealing platform for accelerating I/O intensive network applications. They have been increasingly deployed in data centers and public clouds [33, 66], e.g., in each MS Azure server, enabling line-rate throughput and low, predictable latency at high power efficiency [33]. Many hardware vendors, including Intel, have already announced F-NICs in their future offerings [98].

Data-center F-NICs are used mainly to accelerate infrastructure tasks, such as network functions [61, 72, 80, 119] and software-defined networking [33, 66]. These tasks leverage the F-NIC's *inline processing* capabilities, where data is processed while being transferred between the host and the network, without CPU involvement. In addition, F-NICs are often repurposed as standalone FPGAs running complete applications, e.g., search or deep learning [20, 24, 60, 86].

This paper explores new acceleration opportunities emerging from the growing deployment of F-NICs in clouds, beyond infrastructure tasks and monolithic applications. We seek to leverage F-NICs for inline acceleration of data plane processing in network-intensive applications. For example, F-NICs may run datacenter tax tasks, such as deserialization, hashing, and authentication, which reportedly consume over a quarter of the CPU cycles in data centers [48]. An F-NIC may serve as an extra caching layer for key-value stores, responding directly in case of a hit and eliminating the CPU involvement. We show, for example, that this architecture achieves near line-rate throughput (40 Gbps) for stock `memcached` (§6.2.1). Promising results for application-specific traffic steering, packet transformation, and network stack offloading have been reported in prior work [50, 83]. We discuss these and other applications in §3.

Unfortunately, building such F-NIC-accelerated applications today is hard. First, there are **no adequate operating system abstractions** for inline acceleration of general-purpose applications on F-NICs. Such abstractions should associate F-NIC tasks with the application process, and they should provide well-defined execution boundaries and isolated per-task state while supporting easy integration of F-NIC functionality with the application logic. OpenCL and CUDA provide general lookaside acceleration support, but they are a poor match for F-NICs because they require explicit kernel invocation and data transfers that are irrelevant for the inline processing scenario. Floem [83] provides language-level constructs to accelerate applications on SmartNICs, but it targets CPU-based rather than FPGA-based SmartNIC architectures, and requires application refactoring to use its data

flow model, complicating acceleration of legacy workloads. SmartNIC-accelerated networking frameworks such as AccelNet, eBPF-XDP, and DPDK rte_security [33, 39, 84] are domain specific and lack application-level abstractions. Systems for data plane acceleration, e.g. P4 and FlexNIC [13, 50], expose packet-level match-action rules for F-NIC management, but lack abstractions for application-level semantics.

Second, F-NICs provide **no virtualization support**, preventing their sharing among cloud tenants. Existing virtualization mechanisms for FPGAs [18, 23, 34, 51, 101, 118] and GPUs [49, 73] rely on space partitioning or coarse-grain time sharing of the compute fabric. The former, however, results in hardware underutilization [51, 112, 116], whereas the latter may affect processing latency due to slow context switching and FPGA reconfiguration times [49, 51, 73, 91], making it unsuitable for latency-sensitive tasks. More fundamentally, F-NICs lack *I/O path virtualization* to isolate and protect per-application I/O across shared buses between the network, the FPGA and the host CPU. Thus, current F-NICs cannot guarantee performance isolation for co-located applications.

We introduce **NICA**, a system for FPGA-based **NIC** Server **A**cceleration. NICA introduces new software abstractions and co-designed F-NIC hardware runtime for application acceleration in cloud systems. NICA manages one or more *Accelerator Functional Units (AFUs)* [42, 101] – application-specific hardware accelerators hosted on an F-NIC. Such AFUs can be developed by users, or provided by cloud vendors and deployed on-demand.

OS abstraction. We introduce a novel *ikernel* (inline kernel) abstraction, which represents an AFU in a user program. An application dynamically *attaches* the ikernel to one or more transport layer sockets, activating the respective AFU. Subsequently, all traffic sent and received via these sockets is processed by the AFU without CPU invocation. To communicate via the sockets, the CPU may use standard POSIX sockets API calls, or a high-performance zero-copy interface for application-level messages. The ikernel abstraction is private to a process and provides protection for the AFU application and network state. We discuss the ikernel abstraction, its network stack integration, and FPGA runtime support in §4.1.

AFU virtualization. NICA supports sharing of AFUs among multiple virtual machines (VMs) while guaranteeing state protection and quality of service (QoS). We address two primary requirements: (1) *AFU I/O channel virtualization*, including host and network traffic, by adding anti-spoofing, classification, and packet schedulers for the I/O sent and received by AFUs; and (2) *fine-grain AFU time-sharing*, which uses a hardware task scheduler that switches contexts at a fine granularity, thus allowing better hardware utilization for latency-sensitive applications. We describe AFU virtualization in §4.2 and show how it enables performance isolation in §6.

NICA provides necessary on-FPGA services for accelerating applications on F-NICs in a multi-tenant setting, including an FPGA-resident network transport layer, compute and I/O scheduling blocks, and AFU state isolation. However, the development of high-throughput network-focused AFUs on FPGAs is beyond the scope of this paper. Fortunately, some promising solutions are emerging, such as template libraries with optimized building blocks for network processing [32]. In addition, we believe that cloud providers will increasingly offer AFUs using an "app marketplace" deployment model [4, 17, 40], with a variety of AFUs ready to be used on their infrastructure (see §3).

We prototype NICA[1] on Mellanox Innova F-NICs [68] with a Xilinx FPGA and 2GB of onboard memory. We implement the ikernel API, integrate it with the VMA kernel-bypass network stack [69], and implement the AFU virtualization support in the KVM hypervisor. We also co-design the FPGA hardware support for the software abstractions and AFU virtualization, and we integrate full UDP and partial TCP layer implementation in FPGA.

We evaluate the system with microbenchmarks and accelerate two real-world applications: a `memcached` server and a Node.js-based IoT monitoring server, by implementing the respective AFUs on the F-NIC. Enabling F-NIC acceleration required minimal software changes: 107 additional lines of C and 20 additional lines of JavaScript respectively.

A transparent hot-item cache AFU integrated with `memcached` serves GET hits at 6 µs 99th-percentile latency and 40.3 Mtps throughput for 16B keys/values, 99% of the 40 Gbps line rate and 21.6× faster than the 6-core CPU baseline. For a Zipf(0.99)-distributed workload with 0.2% SETs, NICA acceleration results in a 4.6× speedup.

NICA allows sharing of an F-NIC among multiple VMs while providing significant performance gains. It introduces negligible throughput and latency overheads while maintaining a fair bandwidth allocation, controllable by the hypervisor.

In summary, we make the following contributions:

- We introduce an ikernel OS abstraction for inline acceleration of applications on F-NICs.
- We design an F-NIC virtualization framework that supports I/O QoS and low-latency time sharing of compute resources.
- We implement NICA for Mellanox F-NICs, analyze its performance, and demonstrate the development simplicity and performance benefits for accelerating `memcached` and a Node.js-based IoT server.

2 Background

We describe the F-NIC architecture and survey FPGA programming principles and sharing mechanisms.

2.1 F-NIC architecture

We describe bump-in-the-wire F-NICs, focusing on Mellanox Innova, but others [19, 80, 89] are similar.

[1] https://github.com/acsl-technion/nica

Figure 1: A bump-in-the-wire F-NIC

Bump-in-the-wire. A typical F-NIC (Figure 1) combines a commodity network ASIC (e.g., ConnectX-4 Lx NIC) with an FPGA and local DRAM. The FPGA is located *between* the ASIC and the network port, interposing on all Ethernet traffic in and out of the NIC. The FPGA and the ASIC communicate directly via an internal bus (e.g., 40 Gbps Ethernet), and a PCIe bus connects the ASIC to the host.

The bump-in-the-wire design reuses the existing data and control planes between the CPU and the NIC ASIC, with its QoS management, and virtualization support (SR-IOV), mature DMA engines, and software stack.

F-NIC programming. The development of an F-NIC-accelerated application involves both hardware logic on FPGA and associated software on the CPU. F-NIC vendors provide a lightweight *shell IP*: a set of low-level hardware interfaces for basic operations, including link-layer packet exchange with the network and the host, onboard DRAM access, and control register access. However, the vendor SDK leaves it to customers to implement higher level features such as FPGA network stack processing or virtualization support.

2.2 FPGA concepts

Field Programmable Gate Arrays (FPGAs) are "a sea" of logic, arithmetic, and memory elements, which users can configure to implement custom compute circuits. FPGA compute capacity is determined by the *area* available for the circuits.

FPGA development. FPGAs can be seen as "software-defined" hardware. The software definition, *a design*, is implemented using register transfer languages (RTL) such as Verilog. Additionally, designers can use high-level synthesis (HLS) tools to generate RTL, e.g., from a restricted version of C++ [67]. However, HLS C++ programs are different from CPU programs, and must follow certain rules, including explicit exposure of fine-grain pipeline- and task- parallelism to achieve high performance. Implementation tools then compile the design into an *FPGA image* targeting specific hardware.

Finally, users can load the image onto an FPGA (slow, up to a few seconds), entirely replacing the previous design. Some FPGAs support *partial reconfiguration* to replace only a subset of the entire FPGA, a much faster process (milliseconds), which unfortunately incurs significant area overheads [51].

FPGA sharing. There are three ways to share an FPGA: space partitioning, coarse-grain, and fine-grain time sharing.

Space partitioning divides FPGA resources into disjoint sets used by different AFUs [18, 20, 51]. If shared I/O interfaces (memory, PCIe bus) are securely isolated and multiplexed, this method enables low-overhead FPGA sharing among mutually distrustful AFUs but requires larger FPGAs to fit them all. *Coarse-grain time sharing* dynamically switches AFUs via full or partial reconfiguration [20, 51]. It incurs high switching latency and thus is not suitable for F-NICs' latency-sensitive applications. *Fine-grain time sharing* allows multiple CPU applications to use the same AFU [44]. The AFU implements the context switch internally, in hardware. Packet processing applications such as AccelNet [33] use this approach to process each packet in the context of its associated flow. Such AFUs oversee switching between the contexts; therefore this type of sharing requires AFUs to be *trusted* to ensure fair use and state isolation between their users.

NICA combines both space sharing for untrusted AFUs, and fine-grain time sharing for trusted AFUs, to achieve maximum utilization under area constraints of F-NICs.

3 Motivation

We consider emerging opportunities for application acceleration by using F-NICs in clouds.

3.1 F-NICs in data centers

Microsoft has been among the first to deploy F-NICs at large scale, having installed the Catapult F-NICs in over a million Azure servers. Their recent work [33] analyzes the cost, power, and performance trade-offs of F-NICs in data centers and decisively shows their benefits. Following Azure, other data centers, such as China Mobile [115], Tencent [66], Huawei [88], and Selectel [94], are deploying F-NICs, and leading hardware vendors are adding F-NICs to their offerings [98]. These technology trends suggest that F-NICs will become a commodity, motivating our goal to broaden the scope of their applications.

3.2 Use cases for F-NIC acceleration

What sets F-NICs apart from stand-alone FPGAs is their ability to *interpose and process the network traffic* to and from the host with low overhead. For application acceleration, the application data plane can be partitioned between the F-NIC and the CPU, even for latency-sensitive fine-grain tasks.

We identify several common task categories in the server data plane that benefit from F-NIC acceleration.

Filtering. F-NICs may execute compute-intensive processing, such as per-message stateless authentication (e.g., JSON web token validation [47]), and filter invalid requests before reaching the CPU. We evaluate this example in §6.2.2.

Such filtering patterns arise in many server applications. For example, F-NICs may implement high-performance, simplified versions of popular services to accelerate common behavior (fast path), falling back to the CPU for corner cases (slow path). We show in §6.2.1 how an F-NIC-hosted key-value store cache reduces server load.

Transformation. F-NICs may convert data formats, perform (de)serialization, compression, encryption, or similar datacenter tax tasks [48]. They can change data layout, e.g., transpose matrices [35], sample, or realign data [2, 38] for efficient CPU/GPU processing or storage. As F-NICs may run a (potentially limited) network transport layer, they may speed up CPU transport layer processing [50], as we show in §6.

Transformation is often combined with filtering. For example, to accelerate the log-structured merge (LSM)-trees [22, 79], the F-NIC may store the tree's first level in its local memory, executing updates without interrupting the CPU, batching and sorting them before sending them to the host.

Steering. F-NICs may improve server performance using application-specific packet steering and inter-core load balancing [50, 89], processing complex steering policies at line-rate, e.g., using heavy-hitter approximation sketches [62].

Generation. Applications may offload the transmission of outgoing messages to multiple destinations. Examples include data replication and erasure coding in storage systems [38, 53, 77], and the shuffle stage in distributed analytics engines.

3.3 AFUs in the cloud

AFUs are custom accelerators that can be instantiated on any compatible FPGA and used via a companion software library. There are two deployment models for cloud AFUs: in the FPGA-as-a-Service (FaaS) model, tenants use their own AFUs on cloud infrastructure [3, 5, 41], whereas with the *app marketplace* model, cloud providers offer common AFUs for on-demand deployment [40].

For example, while Amazon provides FaaS, its Marketplace offers third-party AFUs [4]. Similarly, Microsoft deploys its own cloud hardware microservices [17, 24, 33]. The marketplace model opens more opportunities for better F-NIC utilization. As cloud providers develop or audit these AFUs, they can trust them to allow fine-grain sharing. By co-locating tenants that request the same AFU, cloud providers may increase their infrastructure utilization, thereby increasing power efficiency [108] and reducing costs. Pre-designed AFUs are less flexible than customer-provided AFUs, but vendors can offer them at a lower cost due to the more aggressive sharing.

NICA's design supports both deployment models.

4 Design

NICA overview. Figure 2 shows the main NICA components with a single physical AFU. NICA comprises three layers: application-visible OS abstractions and services inside a VM integrated with the network stack (§4.1); the hypervisor layer for managing F-NIC resources and QoS (§4.2); the hardware layer which includes the support for OS abstractions, physical AFU logic (pAFU), a virtualization framework exposing virtual AFUs (vAFUs), and a hardware runtime with network I/O services for application-level message processing on AFUs.

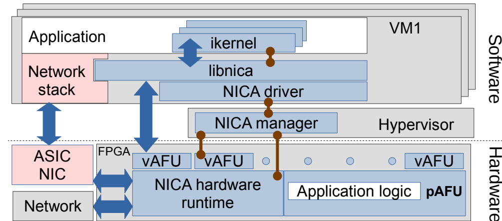


Figure 2: NICA overview. Light blue shapes are NICA components. Blue and brown lines are the data and control path respectively. vAFU: virtual AFU, pAFU: physical AFU.

4.1 Abstractions for inline acceleration

Among our primary goals is to simplify the use of inline accelerators in existing applications with minimal changes. Our abstractions thus provide a general interface for AFU management, which is compatible with standard network I/O interfaces. They allow application control of AFU execution and efficient communication between the host and the AFU.

4.1.1 The ikernel abstraction

An *ikernel* is an OS object that represents an AFU in a user program. An owner process creates an ikernel and controls it exclusively. Essentially, the ikernel extends the process abstraction into the AFU, and NICA protects the ikernel state from other CPU processes and ikernels.

To invoke an AFU, it must be associated with an active network flow. Thus, applications activate the AFU by *attaching* one or more sockets to its ikernel, thereby rerouting the associated traffic through the AFU. The ikernel stops processing the socket's traffic when the application detaches or closes the socket, keeping the AFU state intact for later invocations. Dynamic attachment adds flexibility by enabling software involvement in connection establishment and session preparation, thereby allowing applications to activate an ikernel only for specific clients or request types, for example.

The attachment semantics depends on the protocol. For a UDP socket, the ikernel receives all incoming packets destined to the socket's listening port. For TCP sockets, the effect of attachment depends on the socket state. Attaching a *connected* TCP socket migrates its state to the AFU hardware network layer. After a process attaches a listening TCP socket to an ikernel, the AFU handles new connection requests, as applications with a high connection rate may benefit from accelerating the connection establishment process. Nevertheless, NICA notifies the host network stack about new connections, off the critical path, to provide application control over these connections from the host.

A process may create several shared-nothing ikernels of the same AFU, e.g., to keep different cryptographic contexts for a crypto-AFU, but our intended usage is one ikernel per AFU per process. Multiple threads of a process may attach

Function	Purpose
`ikernel* ik_create(uuid_t, int dram_size)`	Allocate an ikernel
`void ik_destroy(ikernel*)`	Deallocate an ikernel
`int ik_attach(ikernel*, int sock)`	Attach ikernel and socket
`int ik_detach(ikernel*, int sock)`	Detach ikernel and socket
`int ik_command(ikernel*, cmd* desc)`	Invoke RPC command
`cr* ik_create_cr(ikernel*)`	Allocate a ring
`void ik_destroy_cr(cr*)`	Deallocate a ring
`int cr_post_recv/send(cr*, buf*)`	Pass buffers to the ikernel
`int cr_poll(cr*, wc*, int n)`	Poll ring for completion

Table 1: Control (top) and data plane (bottom) ikernel API.

their sockets to the same ikernel, thereby sharing the AFU state among them.

For now, a socket can be attached only to a single ikernel, but we plan to enable ikernel chaining in the future (§4.4).

Alternatives. We choose the ikernel abstraction because it captures the intuitive application-level semantics of inline network processing. We also considered using match-action rules, as in FlexNIC [50] and DPDK [84]. These are not associated with sockets, but rather with packet header rules, e.g., selecting packets of a specific five-tuple. Such an interface suits packet processing but is too low-level for application logic offloading. The ikernel socket-level abstraction hides the details of the hardware-resident network stack and allows simpler integration with existing applications.

4.1.2 Control plane

The control APIs (Table 1) allow initialization, teardown, and access to the AFU-resident application state. Under the hood, they interact with the network stack on the host and on the F-NIC to coordinate resource allocation and AFU processing.

Initialization and attachment. An `ik_create` call initializes an ikernel given an AFU's UUID. When the ikernel attaches to a socket it updates the F-NIC network stack. Once the ikernel is attached, the NICA driver tracks the socket state, detaching the flow when the socket is closed. The application may also detach an ikernel before the connection terminates.

The `ik_create` call may initialize a pre-loaded AFU or load it at runtime using partial reconfiguration. The ikernel abstraction hides the AFU hardware initialization details from the user, leaving the OS in charge of manipulating the FPGA-AFU allocation, similarly to AmorphOS [51].

Application state. Applications may access the ikernel state in the AFU. The hardware could expose the state in two ways: (1) as shared memory between the host and the F-NIC; or (2) using remote procedure calls (RPC) from the CPU to the AFU that retrieve/set the state. Shared memory might not be efficient, however. First, FPGA logic can keep frequently-accessed data in private memory, such as registers or block RAM, for efficiency. This memory is not exposed to the CPU. Further, access to the shared state requires explicit synchronization, which is costly over PCIe. Therefore, we chose the RPC model, which allows the AFU to implement arbitrary atomic transactions, including e.g., getting a snapshot of its state. Internally, NICA also uses the same mechanism to control transport layer and QoS parameters.

Error handling. An AFU that encounters an error exposes it to an ikernel runtime which periodically checks for errors via the RPC mechanism. In addition, the ikernel may abort the connection or detach itself from the respective sockets and forward packets without offloading.

4.1.3 Data plane

NICA provides two ways to perform network I/O with inline acceleration: POSIX API and *custom rings*.

POSIX networking API. After attaching an ikernel to a socket, the application may use standard I/O calls, e.g., `send`, `recv`, and `epoll_wait`, while the AFU transparently processes the data in-flight. We currently support the POSIX APIs only for UDP sockets.

Custom rings. POSIX I/O interfaces incur the overhead of extra data copies into user buffers [81] and host-side network stack processing. On the other hand, an AFU may need to exchange *application-level* messages with the host application. For example, a deserialization AFU may send ready-to-use data objects to the application. Furthermore, an AFU may need to steer the processed messages to different CPU cores, i.e., for application-aware load balancing.

NICA introduces a *custom ring*[2] (CR) abstraction that provides a zero-copy API for sending/receiving *application messages*, bypassing the host network stack. Each ikernel may create multiple associated CRs to enable message steering for multi-core systems.

The CR interfaces are similar to VIA/RDMA verbs [30] (Table 1). Specifically, each CR comprises a queue pair (QP) and a completion queue (CQ). The application allocates its communication memory buffers and registers them with the CR. It then posts the send/receive requests to the respective queue in the QP. The request completions show up in the CQ.

Custom rings vs. random access. FPGA acceleration frameworks [37, 43, 101] and some I/O intensive AFUs [29, 38, 60] allow random access to CPU memory from the AFU, which is useful for fine-grain sharing of data-structures between the CPU and the AFU. NICA currently focuses on the AFU tasks that communicate with the CPU via a streaming I/O pattern, which is much easier to implement using a producer-consumer CR interface. We leave support for random host memory access for future work.

Synchronization. In the most common application scenarios, networking or custom ring operations implicitly synchronize the CPU application and the AFU processing. In more complex cases, when the AFU accumulates the application state (e.g., for network I/O monitoring or consensus), the ikernel

[2] The hardware uses a descriptor *ring* buffer just like a regular NIC, but the buffer contents are application messages rather than raw packets.

RPC interface allows AFU developers to provide application-specific mechanisms to safely access ikernel state.

4.1.4 Usability

We expect adding ikernels to existing applications to require relatively small design or code changes. In case of filtering (see §3.2), an application may still use POSIX sockets as before, while receiving only the filtered data. For example, `memcached` requires no changes to its data processing to use the KVS cache AFU (§6). Data transformation tasks, such as deserialization, may use custom rings to obtain or send back the data in an application-friendly form. Steering applications may use per-core custom rings to get the contents directly to the correct application thread or a GPU. A generation application, e.g., replication, may send only one data copy via the custom ring, while the AFU will distribute it to pre-configured destinations.

4.2 Virtualization

To support fine-grain sharing of AFUs, as required for low latency applications, we introduce the notion of a *virtual AFU, vAFU*, which represents a single isolated hardware entity on the F-NIC. Each vAFU provides state protection and performance isolation across all the shared resources on the F-NIC. To clarify, a vAFU is a hardware entity, whereas an ikernel is an OS object that belongs to a process. Connecting multiple ikernels to the same vAFU might be possible, i.e., allowing in-VM resource allocation policy enforcement, yet we do not support it in our prototype.

One F-NIC may host multiple physical AFUs via space sharing, whereas each such AFU may support multiple vAFUs via fine-grain time sharing, as explained below. For example, our key-value-store cache AFU supports 64 vAFUs, allowing concurrent acceleration of up to 64 different `memcached` servers on the same F-NIC (§6).

Fine-grain AFU sharing. Supporting multiple vAFUs on a single physical AFU requires low-overhead hardware context switching mechanism. The vAFU context includes the ikernel state in DRAM and registers and the contexts of the sockets connected via that vAFU. Each received packet may belong to a different vAFU so slow context switch would not only increase application latency but also increase the required NICA internal buffer space.

To support fine-grain sharing, we store the vAFU context by reserving fast memory for each vAFU rather than evict/reload it to/from slow DRAM memory. Specifically, the AFU registers are replicated to store data for all concurrently active vAFU contexts. Each vAFU is associated with a hypervisor-chosen tag. The AFU switches to the context requested by the scheduler by updating the active tag register. Such a context switch can be extremely fast, e.g., up to 3 clock cycles in our prototype.

However, the number of vAFUs that can be supported is constrained due to the limited size of fast memory on the FPGA. For more vAFUs, AFUs may use DRAM to store the contexts and use latency hiding techniques, i.e., increased concurrency. Our current prototype uses fast memory, yet it is enough to host up to 64 vAFUs for the evaluated applications.

4.2.1 State protection

NICA protects the vAFU state in DRAM, fast memory, and hardware registers. For the DRAM, we use a segment-based MMU for simplicity. Similarly, we protect the control registers of the RPC interface by including a vAFU tag.

Additionally, NICA ensures correct steering of network traffic to and from the vAFU via its on-NIC network stack (§5.3). In particular, it guarantees that a vAFU will not perform network spoofing attacks toward the host and will receive only the packets destined to that vAFU. These two aspects are essential for supporting untrusted AFUs in NICA.

4.2.2 Performance isolation

NICA supports isolation of I/O channels and compute resources. The compute scheduling is necessary only among the vAFUs of the same physical AFU. The FPGA loads different physical AFUs into different partitions, and thus they do not share FPGA compute resources. DRAM bandwidth partitioning is left for future work.

I/O bandwidth sharing. The bandwidth allocation between tenants is often implemented inside a virtual switch or in the NIC internal switch. However, in a bump-in-the-wire architecture the F-NIC sends vAFU-generated messages directly to the network, bypassing these policies. Therefore, NICA provides its own bandwidth allocation mechanisms, similar to the traffic class (TC) mechanisms used in NICs [10].

To control the vAFU egress bandwidth, both towards the CPU and towards the network, we add a set of TC queues (see Figure 3). Packets are classified to these queues and scheduled. We use a work-conserving deficit round robin (DRR) scheduler [95] to allocate bandwidth, but more complex policies can be used. NICA's bandwidth scheduler is trusted and used by all the vAFUs on the F-NIC.

The vAFU recognizes when the TC queues are full and may drop the packets or propagate the contention if possible. For example, it may slow down the host by using custom ring flow control or slow down the sender through explicit congestion notification (ECN).

NICA does not manage the ingress bandwidth into the vAFU from the network or the host, as the sender (TOR or host virtual switch) already shapes ingress traffic.

AFU compute sharing. An AFU must determine which vAFU to activate at any given time, and which packets to serve first. We considered two design options: a general compute scheduler for all AFUs (similar to the I/O scheduler)

or an internal AFU-specific scheduler for each AFU. These two options represent an inherent trade-off between FPGA resource consumption and design generality.

A generic scheduler in front of the vAFUs could reorder packets according to a global policy, simplifying the AFU design. However, such a scheduler requires deep input queues, therefore increasing consumption of F-NIC fast memory. Further, the need for queuing is protocol-dependent. For example, TCP has its own input queues to receive out-of-order packets, so extra scheduling queues would be wasteful. Moreover, AFUs may customize queue contents to save resources, e.g. by keeping parsed requests instead of full packets.

We thus decided to implement a custom, application-specific scheduler in each AFU.

4.3 AFU development

AFUs implement hardware interfaces to receive/transmit transport layer and custom ring data, configuration and control interfaces for RPC, and, optionally, provide vAFU scheduling and virtualization.

All the packets passing through an AFU are tagged with metadata that identifies the associated ikernel and flow, which can be used by the AFU for ikernel state isolation. The AFU receives per-TC usage levels and CR flow control (see §5.2).

While designing such FPGA hardware can be difficult, we try to simplify the development by using high-level synthesis to design our AFUs in C++, and use the `ntl` class library [32] to implement common modules such as AFU schedulers and control-plane interfaces. In addition, the NICA hardware runtime handles some common tasks such as transport processing and egress scheduling, thus simplifying AFU development.

4.4 Discussion

F-NIC transport layer. An inline AFU requires transport layer services to process data at the application layer; it may terminate flows or generate and send new messages. Our current design uses a full implementation of UDP and TCP logic in hardware. With this solution, the F-NIC effectively runs its own complete network stack.

A complete TCP/IP stack in hardware simplifies AFU development but increases F-NIC resource consumption and maintenance difficulty [75]. To eliminate NIC transmission buffers, an AFU could generate retransmissions on-demand or use host memory [85, 97]. If packet reordering is rare, an AFU may process received data only in-order, deferring out-of-order packets to the CPU [85]. A resource-efficient TCP design for inline AFUs warrants further research, so we choose a simple solution to evaluate the ikernel abstraction compatibility with TCP.

Virtual switch offloading. F-NICs intercept the inbound network traffic *before* it reaches the CPU. As a result, it becomes difficult to handle hypervisor policy and virtual networking rules, e.g., as in Open vSwitch, because they are typically handled by the hypervisor's virtual switch software running on CPU. This issue is not unique to NICA and exists with standard SR-IOV NICs [33]. Typical solutions pass the first packet to software and offload per-flow policy to hardware match-action rules [33, 59, 78]. While this may take significant area of the F-NIC's FPGA [33], future F-NIC designs may be able to harden this functionality [20, 31].

Multi-AFU support and services. Our design provides all the necessary mechanisms to run multiple AFUs on the F-NIC: packet schedulers, steering, RPC and MMU isolation modules. Currently, a single socket may only be attached to a single AFU. However, there are use cases for chaining several AFUs in a single application to accelerate various aspects of the server's traffic [16, 56, 117]. Multi-AFU chaining requires extensions to resource isolation mechanisms and software interfaces, which we plan to explore in the future.

5 Implementation

We implement NICA for the Mellanox Innova F-NIC and integrate it with the KVM/QEMU hypervisor and VMA user-space networking library [69].

5.1 AFU virtualization

NICA implements hardware virtualization of the physical AFUs, exposing virtual AFUs (vAFUs in Figure 2) to VMs. Currently, the hypervisor allocates one vAFU for each requested ikernel. NICA isolates the vAFU I/O channels in hardware and requires no software mediation.

We utilize the NIC's SR-IOV functionality to virtualize the data path (both POSIX and custom rings). SR-IOV enables unmediated overhead-free access from the guest to the NIC hardware. In general, implementing SR-IOV in custom accelerators is quite challenging, but the bump-in-the-wire architecture of our F-NIC allows reusing the existing NIC hardware SR-IOV mechanism. For the control plane, which is less sensitive to performance, NICA uses para-virtualization.

5.2 Software

We implement the NICA API in the `libnica` library. It integrates with the VMA user-space networking library, providing the POSIX socket API with kernel bypass and direct hardware access. We modify VMA to support the ikernel abstraction.

The NICA VM driver mediates between `libnica` and the hypervisor's NICA manager daemon, using a para-virtual device (virtio-serial). The NICA manager runs in the hypervisor and controls AFU hardware through the F-NIC kernel driver. NICA software stack is about 2,200 LOC.

Custom ring using RoCE. We use the F-NIC's RoCE support [105] to implement the CR, employing the ASIC NIC

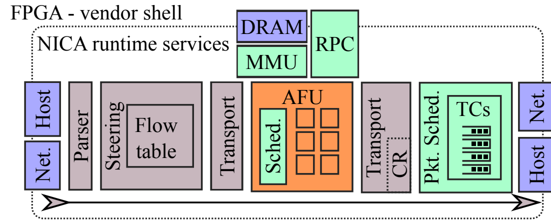


Figure 3: NICA hardware runtime (only ingress is shown, 1 AFU). Isolation modules are green. Each AFU supports multiple vAFUs. Sched.=Scheduler.

hardware and software layers using the bump-in-the-wire architecture. The implementation associates CRs with RoCE unreliable connected (UC) queue pairs (QPs). To send to a specific CR, NICA's transport layer generates RoCE packets to the host, targeting the appropriate QP. The ASIC RoCE engine writes the data directly to the application buffers, providing address translation, DMA, and completion notifications.

In our bump-in-the-wire F-NIC, the FPGA logic does not have a direct end-to-end flow control mechanism with the host, and UC does not provide such a mechanism either. Therefore, NICA adds a credit-based flow control mechanism between the AFU and the CPU application. The custom ring APIs transparently invoke this mechanism.

5.3 Hardware runtime

Figure 3 shows our FPGA processing pipeline. For clarity, we describe ingress (from network to host) only. The FPGA runtime provides the hardware support for inline programming abstractions and the essential services for inline acceleration. These include: (1) the custom rings and RPC mechanism to support efficient data and control plane primitives for ikernels; (2) a memory management unit (MMU) for memory isolation; (3) a network processing stack to support application-level processing in the AFU, which includes the parser, flow steering, and the transport layer; and (4) a virtualization layer, implementing AFU and packet schedulers.

We develop NICA and the evaluated AFUs in HLS [114] and Verilog. Table 2 shows the FPGA resources and number of code lines. NICA operates the FPGA at 216.25 MHz.

TCP/IP implementation. Our prototype includes full support for UDP and partial support for TCP. The UDP/IP service splits/combines the header and the payload. As the CR utilizes RoCE over UDP, it also uses the UDP/IP service.

The TCP implementation builds on an existing 10 Gbps FPGA TCP/IP stack [96]. Its integration with NICA is incomplete, as it lacks virtualization and socket migration support (though existing techniques apply [8, 27]). It is included primarily to validate how NICA abstractions hold with TCP.

Table 2: FPGA utilization and lines of code. LUTs: lookup tables, FFs: flip-flops, RAMB18: block RAM units.

	Module	Area (% of total) LUTs	FFs	RAMB18	LOC HLS	Verilog
System	NICA	13%	9%	13%	6643	1736
	TCP stack	6%	4%	13%	15303	1110
	Vendor shell	51%	32%	7%		
Apps	NICA-KVcache	5%	2%	2%	975	
	IoT server	10%	7%	8%	646	1627

5.4 Limitations

Our prototype may run only two physical AFUs, where one is a minimal AFU that passes through unmodified traffic. This is not a design limitation but stems from the FPGA area constraints (see Table 2). Further, NICA does not yet support virtual switch offloading, and our current CR implementation does not transmit, only receives. In addition, our F-NIC does not support partial reconfiguration. We hope the next generation of the F-NIC [31] will resolve these limitations, as it is expected to have a larger FPGA with more space and hardened network virtualization support.

NICA performance drops dramatically when crossing NUMA links. We are investigating a potential hardware bug.

6 Evaluation

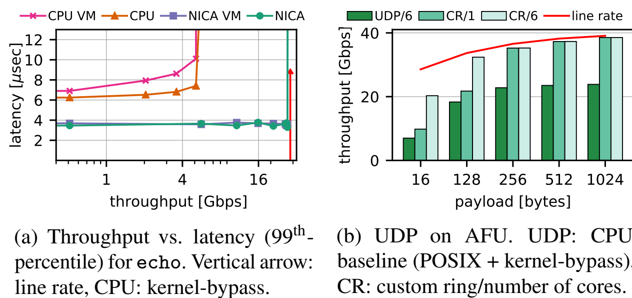
Hardware setup. We use four machines with Intel® Xeon® E5-2620 v2 2.1 GHz CPUs, connected via a Mellanox SN2100 40 Gbps switch. Three (clients) use Mellanox ConnectX®-4 Lx EN NIC, and one (server) uses a 40 Gbps Mellanox® Innova™ Flex 4 Lx EN (1st gen.) F-NIC, equipped with a Xilinx XCKU060 FPGA. The server is a dual socket NUMA machine with 64 GB RAM. Hyper-threading and power saving settings are disabled.

CPU baseline. We use VMA [69] user-level network stack with kernel bypass, optimized by Mellanox and broadly used for high-performance networking [33]. We use commodity NICs with the same ASIC as our F-NIC but without the FPGA. Due to the NUMA performance issue of the current prototype (§5.4), to allow a fair comparison, we constrain our experiments to the NUMA node closer to the NIC.

F-NIC maximum power consumption. The F-NIC consumes up to 30 W [68] vs. 14.2 W [70] for the client NICs.

Performance measurement. We use `sockperf` [71], a benchmarking tool optimized for VMA. To reliably measure performance, we use performance counters on NICA's FPGA runtime, the NIC, and the switch. We run each experiment 5 times, each 60 second long.

NICA configuration. We set a max. of 4 TCs, 64 ikernels, VMs, and custom rings, 1K UDP ports, and 10K TCP flows.



(a) Throughput vs. latency (99th-percentile) for `echo`. Vertical arrow: line rate, CPU: kernel-bypass.

(b) UDP on AFU. UDP: CPU baseline (POSIX + kernel-bypass), CR: custom ring/number of cores.

Figure 4: Microbenchmarks

6.1 Microbenchmarks

We use several microbenchmarks to evaluate the benefits of NICA acceleration through filtering and transport layer acceleration and to estimate virtualization overheads.

Experiment 1: Virtualization performance. Figure 4a shows the throughput-latency comparison of bare-metal and virtualized echo server AFU vs. the CPU baseline for 64-byte packets. We measure no overheads of the AFU virtualization.

At 5 Gbps, the latency of the virtualized AFU is 2×/2.8× lower than bare-metal/in-VM CPU server respectively. At 6.7 Gbps, the baseline latency spikes to 38 µs, while the AFU achieves up to 27.6 Gbps at 4 µs latency, above which we see packet drops. The stable low latency at high throughput is a valuable property of F-NIC accelerators.

Experiment 2: UDP performance. We run a pass-through AFU that receives UDP packets and transfers them to the host via CRs, saving the host UDP processing. The CPU baseline uses VMA for POSIX API kernel bypass, with 6 CPU cores. Figure 4b shows the throughput for different packet sizes. Offloading UDP processing to the AFU boosts the throughput from 2.9× and 1.7× for small and large packets respectively. For larger packets, a single-core CR outperforms 6-core UDP.

Experiment 3: TCP performance. We evaluate NICA's preliminary TCP support by accelerating a monitoring server microbenchmark. The server receives integers as 18-byte messages (4-byte integers with a 14-byte sockperf header) and computes their average, alerting the user when the received values are above a given threshold. With NICA, the AFU maintains the average and sends only the messages above the threshold via the custom ring (bypassing the host TCP stack).

For 6 flows from 6 clients, the AFU consumes 34.8M messages/sec, 3× faster than the baseline's 11.5M messages/sec (single core). The AFU benefits diminish as the portion of the messages sent to the host increases, down to a modest 11% throughput improvement. This indicates that *the F-NIC transport layer processing contributes much less than filtering to the overall performance benefits*.

Experiment 4: I/O isolation overheads. We evaluate the egress scheduler when using two AFUs: a traffic generator AFU and a pass-through AFU. The former generates messages to the network at maximum throughput. The latter transfers messages between the host and the network. These AFUs share the network egress I/O channel and are assigned to separate traffic classes. We set the scheduler quantum to 1 KB.

We measure the latency of a few 64-byte packets sent via the pass-through AFU while the generator AFU sends 1514-byte packets. At 38.4 Gbps load, the low-latency pass-through packets suffer a 1 µs overhead to 99th-percentile latency compared to an empty system. This result demonstrates that the *I/O isolation mechanism achieves low overhead even under heavy contention*.

6.2 Application benchmarks

We accelerate two large applications: `memcached` and a Node.js-based IoT server. We build a transparent cache AFU for the former and an authentication AFU for the latter, integrating both into the CPU software.

6.2.1 Transparent `memcached` cache

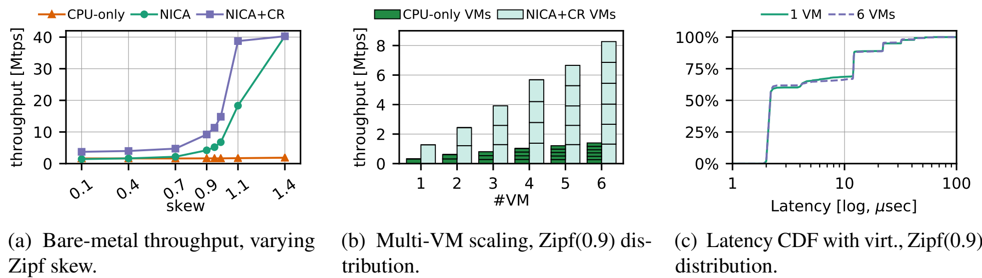
We prototype a transparent look-through cache for `memcached`, called NICA-KVcache. The AFU parses `memcached`'s ASCII UDP protocol and serves GETs directly from its F-NIC DRAM-resident cache. The AFU passes GET misses and other update requests to the host. Upon update, the AFU *invalidates* the respective cache entry. The AFU populates the cache by intercepting GET *responses* from the host, ensuring coherence even if the host drops the updates due to overload. The AFU caches keys/values of up to 16-byte and uses a direct-mapped cache for simplicity.

We implement two designs: one with POSIX API and another with CRs. The former requires changing `memcached` to instantiate the ikernel and attach sockets. The latter introduces CR polling to the `memcached` worker thread event loop. Adding the F-NIC acceleration support required 107 and 135 LOC for the POSIX API and CR versions respectively.

Workload. We initialize the CPU server with 32 M 16-byte keys and values (4 GB with overheads) and set the AFU cache to store 2 M keys per-ikernel (128 MB RAM). The CPU baseline uses an unmodified `memcached` with the VMA network stack. Clients generate a YCSB-like [25] workload with varying skew using `sockperf`.

Bare-metal performance. Figure 5a shows that for lower skews (high miss rate), the CPU (6 cores) is the bottleneck. With Zipf(0.99) distribution (YCSB's default), NICA+CR achieves 9× speedup. For 100% hit-rate, the AFU becomes network-bound (99% of 40 Gbps line-rate), resulting in 21× higher throughput than the baseline.

The cache hit-rate also dictates the latency distribution (not shown). We observe a mixture of two distributions: cache hits and cache misses. With Zipf(0.99) distribution and 1 Mtps load, the F-NIC serves cache hits at a stable 2.1 µs. Misses, served by the host, are 6 µs at the 99th-percentile, versus



(a) Bare-metal throughput, varying Zipf skew.

(b) Multi-VM scaling, Zipf(0.9) distribution.

(c) Latency CDF with virt., Zipf(0.9) distribution.

Figure 5: NICA-KVcache results, CPU+VMA (kernel-bypass) vs. NICA with/without a custom ring (CR).

Table 3: NICA-KVcache throughput [Mtps] with 0.2% SETs.

Skew	0.90	0.95	0.99	1.10
Baseline (CPU-only with kernel-bypass)	1.55	1.55	1.55	1.55
NICA with custom ring	5.98	6.51	7.10	8.28

10.5 µs in the baseline. *The latency improvement is due to the reduced CPU load as a result of filtering.*

Table 3 shows the throughput with 0.2% SETs (common in Facebook [15]). At Zipf(0.99), NICA is 4.6× faster than the baseline. With 10% SETs (not shown), CPU throughput dominates, thus NICA shows no performance improvement.

Other KVS implementations. NICA-KVcache offers significant advantages even when used with highly optimized CPU-only KVS implementations, such as MICA [63], which achieve line-rate throughput using CPU cores alone. In this case, NICA-KVcache reduces the required number of CPU cores by filtering all the cache hits and leaving only the misses to the CPU, thereby improving the overall system efficiency. More specifically, for a given hit rate in the NICA-KVcache, achieving line-rate requires the CPU throughput to be $line_rate \cdot (1 - hit_rate)$ transactions per second.

For example, MICA [63] reaches 5 Mtps on a single CPU core with 100% GETs for 32 M 16-byte keys and values (1GB of data) with a Zipf(0.99) distribution. Optimistically assuming perfect scaling, MICA would reach line-rate (59.5 Mtps) with 12 cores, without NICA-KVcache acceleration. In contrast, with NICA-KVcache of size 128MB, running the same Zipf(0.99) key distribution results in 75% hit-rate, thus the CPU only handles 14.9 Mtps, utilizing just 3 CPU cores.

This result demonstrates that the use of NICA for accelerating key-value stores is cost-effective, considering that a single CPU core is reportedly more expensive than a SmartNIC [33].

Accelerated KVS. Floem [83] implemented a similar key-value store cache on a Cavium SmartNIC and reported a 3.6× performance improvement with 100% hit-rate with write-back, and no benefits for 10% SETs write-through, as in NICA. Rather than `memcached`, Floem required a custom KVS server, however. KV-Direct [60] with small requests achieves comparable performance to NICA (with 100% hits) but reaches 180 Mtps using client-side batching. Unlike NICA-KVcache, its data-plane is fully implemented in hardware, and it only uses the host for slab allocation.

Contribution of network-stack processing. Figure 5a shows that using CR for low cache hit rates results in 2.2× speedup over the CPU baseline. In this case, the use of CR eliminates the network stack processing on the host but keeps the application processing on the CPU. Naturally, higher hit rates result in a higher portion of the requests handled by the AFU, and much higher speedups. This experiment suggests that *the network stack offloading alone is not enough to reach the full performance potential of the F-NIC acceleration*.

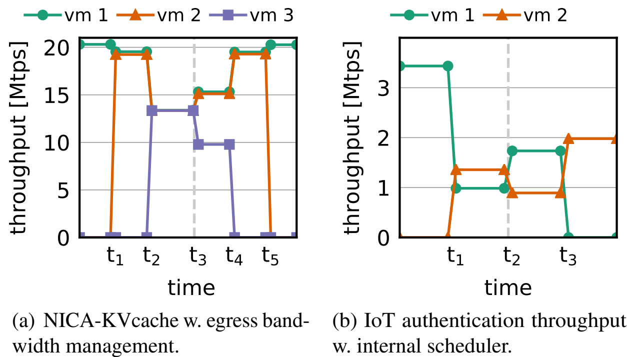
Virtualization performance. We evaluate the performance with a varying number of VMs. Each VM uses 5 GB of server RAM, 1 dedicated CPU core, a vAFU, and 2 M keys worth of vAFU cache. For Zipf(0.9), Figure 5b shows near-linear scaling, consistently achieving a 5.6× speedup over the CPU. Further, we observe no measurable negative impact of virtualization on vAFU latency. The system achieves similar results with 64 M keys per VM, utilizing most of the 64 GB RAM of our machine.

Figure 5c shows the latency distribution of a single VM and 6 VMs executions under 1.3 Mtps load, for a Zipf(0.90) workload. The latency increases for the top 40% of the requests, which matches the expected hit-rate. We observe that the VM CPU latency is much higher than the bare-metal latency reported above, but cache hits are served at the same latency with and without virtualization.

This experiment confirms that *AFU fine-grain sharing is feasible and effective*.

Network bandwidth isolation. We use 3 VMs, associated with 3 TCs, and initially configure them to share the egress bandwidth equally. We use a Zipf(1.4) distribution (99.9% hit-rate), and a 20 Mtps load on each VM, to stress the scheduler.

Figure 6a shows the throughput of each VM over time. At first, only VM 1 is active, using the whole AFU. When VM 2's clients start, the combined egress throughput is barely above the AFU's maximum (39 Mtps), and the clients process 19 Mtps each. When VM 3's clients start, the combined throughput surpasses the maximum, and the scheduler divides the bandwidth equally (13 Mtps per VM). At t_3, we change the bandwidth allocation to 40%/40%/20% and observe an asym-



(a) NICA-KVcache w. egress bandwidth management.

(b) IoT authentication throughput w. internal scheduler.

Figure 6: QoS experiments

Table 4: Node.js goodput (valid req. received) under DoS.

Valid packet ratio	40%	60%	80%	100%
Baseline (req/sec)	1489	2294	3131	3960
NICA (req/sec)	5165 (3.4×)	5165 (2.3×)	5231 (1.6×)	5181 (1.3×)

metric allocation. This confirms the *NICA egress isolation is successful in allocating bandwidth among the tenants*.

6.2.2 IoT authentication

We prototype an IoT monitoring server using Node.js with JSON web token (JWT) stateless authentication. The JavaScript-based server exposes an endpoint to which IoT devices publish their measurement using the CoAP protocol [12], similarly to the SAMSUNG Artik IoT cloud API [92]. The payload of each request contains an authentication token, which includes the device ID and a timestamp, and signed using HMAC-SHA256. Invalid requests are discarded.

Our prototype authentication AFU parses received packets, extracts the token, verifies the signatures (using a SHA-256 accelerator [99]) and drops requests with invalid tokens. Valid requests are passed to the CPU and only undergo token expiration check there.

We evaluate our IoT authentication accelerator against a software-only Node.js server. Adding NICA support using POSIX APIs required 20 JavaScript LOC and 34 lines for the libnica generic Node.js module, demonstrating *the simplicity of integrating the ikernel abstraction with complex software*.

In this experiment, we simulate a Denial of Service (DoS) attack by sending a varying number of invalid tokens with incorrect signatures in the input stream. Table 4 shows the goodput, in requests/sec, as a function of the valid packet ratio. While the baseline degrades linearly, NICA maintains a constant goodput by filtering the invalid packets.

One may wonder whether optimizing the Node.js server (e.g., rewriting it in C) would diminish the AFU acceleration benefits. We argue that this is not the case. The AFU hardware achieves the throughput of 3.5 Mtps, about 3 orders of magnitude higher than the software throughput. As long as the rest of the CPU processing pipeline remains the bottleneck, the AFU remains effective. Additionally, compute acceleration alone results in only 30% speedup. The remaining speedup is due to filtering invalid packets, which would be helpful in the CPU-optimized version too.

AFU compute sharing. The AFU's throughput can be bounded by its SHA-256 hashing units and depends on the input JWT token sizes. To fairly share the hashing units among vAFUs, we introduce a custom DRR scheduler (§4.2.2)) that controls the per-VM utilization of the AFU hashing units.

We use 2 VMs to demonstrate the performance isolation. Clients send 10 Mtps of invalid requests to each VM, but VM 2 receives requests with 40% larger tokens. We start the experiment with the scheduler disabled and enable it mid-run.

Figure 6b shows the throughput of each VM over time. At first, only VM 1 clients are active, allowing the AFU to process at max speed (3.5 Mtps). When VM 2 begins receiving at t_1, VM 1 processes only 28% of the requests, which is below its fair share. With the scheduler enabled, at t_2, both VMs receive half of their respective maximum throughput. We observe that *NICA's compute performance isolation is essential to allow sharing of compute-bound AFUs*.

7 Related work

NIC-based acceleration. Commodity NICs have been offering network stack offloads ranging from checksum calculations, segmentation, and receive-side-scaling (RSS) to RDMA [9, 82, 87, 105, 110] and TCP offload engines [75]. Such offloads are limited to network and transport layer processing, while NICA focuses on the application layer.

Our work builds upon previous attempts to accelerate general purpose applications through inline processing in SmartNICs. Early work on Network Processing Units (NPUs) [1, 113] programming abstractions [16, 56] has shown the potential of customizing the I/O path for applications. More recently, FlexNIC [50] has proposed an RMT-based [14] NIC for inline acceleration of application packet processing, showing how to leverage RMT hardware for application acceleration. Floem [83] aids design of NPU accelerated applications. sPIN [38] offers inline acceleration of high-performance computing (HPC) tasks such as tag-matching, data transformation, or replication, but the Portals 4 host abstraction is unsuitable for socket applications.

While we also consider inline acceleration, our goals, design, platform, and evaluation methodology are different. FlexNIC focuses on applications of SmartNIC RMT acceleration, whereas NICA offers convenient OS abstractions for integrating inline accelerators into user applications. FlexNIC targets RMT SmartNICs with constrained functionality, whereas NICA targets more flexible bump-in-the-wire FPGAs. These may run large parts of application logic, necessitating more expressive interfaces for state and execution management, such as host-NIC network stack interaction. As RMT devices

are designed to work at line-rate, performance isolation of concurrent application pipelines is unnecessary; conversely, we show that QoS support is essential to expose F-NICs in cloud systems.

Packet processing frameworks such as DPDK and eBPF-XDP [39] include inline acceleration mechanisms, e.g., for cryptographic protocols such as IPSec [84] or offloading eBPF programs to SmartNICs [52]. However, these target system-wide packet processing tasks, so they lack a transport layer, network stack integration, and multiple application support.

Linux also supports attaching eBPF programs to sockets [26], similarly to ikernels, to perform inline packet processing. However, such programs cannot process transmitted packets or generate new ones, and use a POSIX API data-path, whereas ikernels enable zero-copy application messaging.

C-CORE [56] proposes the stream handlers abstraction for inline processing, but unlike ikernels, they provide no virtualization mechanisms. Streamline [16] is an OS subsystem for tailoring application I/O path that uses UNIX pipes as an abstraction, but it does not allow dynamic attachment and configuration of filters.

Some F-NIC vendors have proprietary APIs for inline application development. Solarflare AOE allows low latency TCP transmission [97] from an F-NIC. Unlike NICA, it only offloads transmissions. Maxeler MPC-N supports inline UDP/TCP application acceleration [7]. All the above lack virtualization support, and their proprietary host application abstractions are too hardware specific.

SmartNIC applications. Eden [6] and AccelNet [33] accelerate network functions on data-center end-nodes with SmartNICs. However, these are loosely coupled with host applications, whereas NICA's model couples the AFU logic with the host server logic.

Hardware accelerators for Network Function Virtualization (NFV) [18, 34, 117] target the NFV domain and hence do not provide abstractions for general purpose applications, lack host-accelerator network stack integration provided by ikernels, and provide no I/O path virtualization to/from the accelerator.

Several works have accelerated specific applications on F-NICs [24, 57, 60, 64, 106, 107]. NICA provides an infrastructure for building such AFUs in the clouds.

Languages for SmartNIC AFU development. P4 [13] is a DSL for implementing network functions with implementations for FPGAs [100, 111]. The Click [55] router has been ported to F-NICs [61, 90]. Emu [102] enables the development of network functions on NetFPGA using HLS. These can be used to simplify AFU development for NICA, but do not provide application-level abstractions.

Floem [83] is a DSL for NPU-accelerated applications. However, it requires refactoring applications to its DSL, while ikernel abstraction is less intrusive.

FPGA virtualization and sharing. AmorphOS [51] improves FPGA utilization by sharing an FPGA among multiple AFUs, and dynamically switching AFUs. Its hull isolates different AFUs used by different applications. We apply similar mechanisms to F-NIC. However, AmorphOS does not isolate FPGA network interfaces, and its context switching mechanism is not suitable for latency-sensitive networking applications.

Multes [44] shares an FPGA among tenants using a single pipeline. AccelNet [33] allows flow-context switching on a packet-by-packet basis. NICA's fine-grained time-sharing design is similar, but its goal is to virtualize inline accelerators for application layer, rather than a standalone FPGA application or cloud network/transport layers.

Remote/distributed FPGA frameworks [7, 19, 104] share FPGAs over the network with a remote CPU. Other have virtualized local look-aside accelerators [23, 36, 101, 118]. In contrast, NICA virtualizes local inline networking AFUs.

Standalone FPGAs, GPUs, or switches. Our choice of FPGA-based SmartNICs has been motivated by prior works on accelerating networking applications [11, 21, 45, 76, 103, 109]. Unlike NICA, they focus on standalone FPGAs.

Other inline acceleration techniques let GPU kernels control communication using GPU-centric networking abstractions [28, 54, 58, 74], or process data in transit on programmable switches or network accelerators [46, 62, 65, 93]. Conversely, NICA provides tighter integration of server software and AFUs. This simplifies integration with legacy programs and makes acceleration transparent for clients.

8 Conclusion and future work

As F-NICs are becoming common in data centers, new use cases for application layer inline acceleration are starting to emerge. NICA provides the ikernel OS abstraction to easily integrate F-NIC-based accelerators into applications and introduces virtualization mechanisms to share them securely and fairly in cloud systems. NICA's real-world prototype demonstrates the significant performance potential for inline acceleration of virtualized server systems, with minimal software development effort.

We believe NICA's inline abstractions are suitable beyond F-NICs and plan to investigate their use in CPU-FPGA systems and non-FPGA SmartNICs. NICA raises a range of research topics, such as distributed heterogeneous architectures, accelerator chaining, and reliable transport offloading, which we will explore in the future.

Acknowledgments

We thank Chris Rossbach, Michael Swift, Ada Gavrilovska, Aleksandar Dragojevic, and our shepherd Scott Rixner for their valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18), the Israeli Innovation Authority Hiper Consortium, the Technion Hiroshi Fujiwara Cybersecurity center, as well as Mellanox hardware donations and technical support.

References

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, 2002.

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: harnessing data parallel hardware for server workloads. In *ASPLOS '14*. ACM, 2014, pp. 19–34.

[3] Alibaba Cloud. Instance type families: f1, compute optimized type family with FPGA. (Accessed: Jan. 2019). URL: https://www.alibabacloud.com/help/doc-detail/25378.htm%5C#f1.

[4] Amazon. AWS Marketplace – F1 search results. (Accessed: Dec. 2018). URL: https://aws.amazon.com/marketplace/search/results?x=0&y=0&searchTerms=F1&page=1&ref_=nav_search_box.

[5] Amazon Web Services. Amazon EC2 F1 instances. (Accessed: Jan. 2019). 2016. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[6] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling end-host network functions. In *SIGCOMM '15*. ACM, 2015, pp. 493–507.

[7] T. Becker, O. Mencer, S. Weston, and G. Gaydadjiev. Maxeler data-flow in computational finance. In, *FPGA Based Accelerators for Financial Applications*, pp. 243–266. Springer, 2015.

[8] M. Bernaschi, F. Casadei, and P. Tassotti. SockMi: a solution for migrating TCP/IP connections. In *PDP 2007*, Feb. 2007, pp. 221–228.

[9] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® Omni-Path Architecture: enabling scalable, high performance fabrics. In *HOTI 2015*, Aug. 2015, pp. 1–9.

[10] D. L. Black, Z. Wang, M. A. Carlson, W. Weiss, E. B. Davies, and S. L. Blake. An Architecture for Differentiated Services. RFC 2475. Dec. 1998. URL: https://rfc-editor.org/rfc/rfc2475.txt.

[11] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *HotCloud'13*. USENIX, 2013.

[12] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: an application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, Mar. 2012.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[14] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM '13*. ACM, 2013, pp. 99–110.

[15] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC 2013*. USENIX, 2013, pp. 49–60.

[16] W. d. Bruijn, H. Bos, and H. Bal. Application-tailored I/O with Streamline. *ACM Trans. Comput. Syst.*, 29(2):6:1–6:33, May 2011.

[17] D. Burger. Microsoft unveils Project Brainwave for real-time AI. (Accessed: Sep. 2018). 2017. URL: https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[18] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. FPGAs in the cloud: booting virtualized hardware accelerators with OpenStack. In *FCCM 2014*, May 2014, pp. 109–116.

[19] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *MICRO-49*. IEEE Computer Society, Oct. 2016.

[20] A. Caulfield, P. Costa, and M. Ghobadi. Beyond SmartNICs: towards a fully programmable cloud. In *HPSR 2018*, June 2018.

[21] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *FPGA '13*. ACM, 2013, pp. 245–254.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4, 2008.

[23] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling FPGAs in the cloud. In *CF '14*. ACM, 2014, 3:1–3:10.

[24] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. v. Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger. Serving DNNs in real time at datacenter scale with project Brainwave. *IEEE Micro*, 38(2):8–20, Mar. 2018.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*. ACM, 2010, pp. 143–154.

[26] J. Corbet. Attaching eBPF programs to sockets. (Accessed: Jan. 2019). 2014. URL: https://lwn.net/Articles/625224/.

[27] J. Corbet. TCP connection repair. (Accessed: Jan. 2019). 2012. URL: https://lwn.net/Articles/495304/.

[28] F. Daoud, A. Watad, and M. Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *ROSS '16*. ACM, 2016, 6:1–6:8.

[29] A. Dragojević. The configurable cloud: accelerating hyperscale datacenter services with FPGAs. Presented at MARS'17. (Accessed: Jan. 2019). 2017. URL: https://sites.google.com/site/mars2017eurosys/Program/keynotes/MARS%20alekd%20shared.pdf.

[30] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, Mar. 1998.

[31] H. Eran, D. Levi, L. Liss, and M. Silberstein. NFV acceleration: the role of the NIC. In *SFMA'18*, 2018.

[32] H. Eran, L. Zeno, Z. István, and M. Silberstein. Design patterns for code reuse in HLS packet processing pipelines. In *FCCM '19*. IEEE Computer Society, 2019.

[33] 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 *NSDI '18*. USENIX Association, 2018, pp. 51–66.

[34] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. OpenANFV: accelerating network function virtualization with a consolidated framework in openstack. *ACM SIGCOMM Comput. Commun. Rev.*, 44(4):353–354, Aug. 2014.

[35] J. Gomez-Luna, I.-J. Sung, L.-W. Chang, J. M. González-Linares, N. Guil, and W.-M. W. Hwu. In-place matrix transposition on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):776–788, 2016.

[36] L. Gong and X. Zeng. Virtio-crypto: a new framework of cryptography virtio device. KVM Forum. (Accessed: Jan. 2019). 2017. URL: http://events17.linuxfoundation.org/sites/events/files/slides/Introduction%20of%20virtio%20crypto%20device.pdf.

[37] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio. On how to improve FPGA-based systems design productivity via SDAccel. In *IPDPS Workshops 2016*, May 2016, pp. 247–252.

[38] T. Hoefler, S. D. Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance streaming Processing in the Network. In *SC 2017*, Nov. 2017.

[39] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress Data Path: fast programmable packet processing in the operating system kernel. In *CoNEXT '18*. ACM, 2018, pp. 54–66.

[40] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *SoCC '16*. ACM, 2016, pp. 456–469.

[41] Huawei Cloud. FPGA-accelerated cloud server. (Accessed: Jan. 2019). URL: https://www.huaweicloud.com/en-us/product/fcs.html.

[42] Intel. Accelerator functional unit (AFU) developer's guide. (Accessed: Sep. 2018). 2018. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-afu-dev-v1-1.pdf.

[43] Intel. Intel FPGA SDK for OpenCL programming guide. (Accessed: Sep. 2018). 2018. URL: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.

[44] Z. István, G. Alonso, and A. Singla. Providing multi-tenant services with FPGAs: case study on a key-value store. In *FPL 2018*, Aug. 2018, pp. 119–1195.

[45] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: inexpensive coordination in hardware. In *NSDI '16*. USENIX Association, 2016, pp. 425–438.

[46] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: balancing key-value stores with fast in-network caching. In *SOSP '17*. ACM, 2017, pp. 121–136.

[47] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519. May 2015. URL: https://rfc-editor.org/rfc/rfc7519.txt.

[48] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ISCA '15*. ACM, 2015, pp. 158–169.

[49] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC 2011*. USENIX Association, 2011, pp. 2–2.

[50] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with FlexNIC. In *ASPLOS '16*. ACM, 2016, pp. 67–81.

[51] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *OSDI 2018*. USENIX Association, Oct. 2018.

[52] J. Kicinski and N. Viljoen. eBPF hardware offload to SmartNICs: cls_bpf and XDP. In *Netdev 1.2*, 2016.

[53] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM '18*. ACM, 2018, pp. 297–312.

[54] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: networking abstractions for GPU programs. In *OSDI 2014*. USENIX Association, Oct. 2014, pp. 201–216.

[55] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[56] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-CORE: using communication cores for high performance network services. In *NCA 2005*, July 2005, pp. 171–178.

[57] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based in-line accelerator for memcached. *IEEE Comput. Archit. Lett.*, 13(2):57–60, July 2014.

[58] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John. GPU triggered networking for intra-kernel communications. In *SC '17*. ACM, 2017, 22:1–22:12.

[59] I. Lesokhin, H. Eran, and O. Gerlitz. Flow-based tunneling for SR-IOV using switchdev API. In *Netdev 1.1*, Feb. 2016.

[60] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: high-performance in-memory key-value store with programmable NIC. In *SOSP '17*. ACM, 2017, pp. 137–152.

[61] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM '16*. ACM, 2016, pp. 1–14.

[62] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *NSDI '16*. USENIX Association, 2016, pp. 31–44.

[63] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI '14*. USENIX Association, 2014, pp. 429–444.

[64] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ISCA '13*. ACM, 2013, pp. 36–47.

[65] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: toward in-network computation with an in-network cache. In *ASPLOS '17*. ACM, 2017, pp. 795–809.

[66] L. L. Luo. Towards converged SmartNIC architecture for bare metal & public clouds. APNet 2018 (Accessed: Jan. 2019). 2018. URL: https://conferences.sigcomm.org/events/apnet2018/slides/larry.pdf.

[67] G. Martin and G. Smith. High-level synthesis: past, present, and future. *IEEE Des. Test. Comput.*, 26(4):18–25, 2009.

[68] Mellanox Technologies. Innova Flex 4 Lx EN adapter card product brief. (Accessed: Jan. 2019). 2017. URL: https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf.

[69] Mellanox Technologies. libvma: Linux user-space library for network socket acceleration based on RDMA compatible network adaptors. (Accessed: Jan. 2019). 2018. URL: https://github.com/Mellanox/libvma.

[70] Mellanox Technologies. Mellanox Technologies ConnectX®-4 Lx single 40/50 Gb/s Ethernet QSFP28 port adapter card user manual. (Accessed: Jan. 2019). URL: https://www.mellanox.com/related-docs/user_manuals/ConnectX-4_Lx_Single_40_50_Gbs_Ethernet_QSFP28_Port_Adapter_Card_User_Manual.pdf.

[71] Mellanox Technologies. sockperf: network benchmarking utility. (Accessed: Jan. 2019). 2018. URL: https://github.com/Mellanox/sockperf.

[72] Mellanox Technologies. Whitepaper: Mellanox Innova IPSec: achieve groundbreaking security for VPN, data privacy & data-in-motion, while reducing total cost of ownership (TCO). (Accessed: Jan. 2019). 2018. URL: https://www.mellanox.com/related-docs/whitepapers/WP_Innova_IPsec.pdf.

[73] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ASPLOS '14*. ACM, 2014, pp. 301–316.

[74] C. Min, W. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *EuroSys '18*. ACM, 2018, 36:1–36:15.

[75] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *HOTOS'03*. USENIX Association, 2003, pp. 5–5.

[76] R. Müller and K. Eguro. FPGA-accelerated deserialization of object structures. Tech. rep. MSR-TR-2009-126. Microsoft Research Redmond, 2009.

[77] R. Nakhjavani and J. Zhu. A case for common-case: on FPGA acceleration of erasure coding. In *FCCM 2017*, Apr. 2017, pp. 81–81.

[78] Netronome. Agilio OVS firewall software. (Accessed: Jan. 2019). 2017. URL: https://www.netronome.com/media/documents/PB_Agilio_OVS_FW_SW.pdf.

[79] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[80] A. Pant, K. Siva, and N. Tan. IPSec Acceleration: securing your data across the data center. Oracle Open World (Accessed: Jan. 2019). 2017. URL: https://static.rainfocus.com/oracle/oow17/sess/1502318673168001SKY0/PF/OOW%20Technical%20Session%20Final%20100217_1507049724149001WUcf.pdf.

[81] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: the operating system is the control plane. In *OSDI 2014*. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf, Oct. 2014, pp. 1–16.

[82] G. F. Pfister. An introduction to the InfiniBand™ architecture. In, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, part 42. John Wiley & Sons, Inc., 1st ed., 2001.

[83] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: a programming system for NIC-accelerated network applications. In *OSDI 2018*. USENIX Association, Oct. 2018, pp. 663–679.

[84] B. Pismenny, D. Doherty, and H. Agrawal. rte_security: enabling hardware acceleration of security protocols. DPDK Summit Userspace. (Accessed: Jan. 2019). 2017. URL: https://dpdksummit.com/Archive/pdf/2017Userspace/DPDK-Userspace2017-Day1-9-security-presentation.pdf.

[85] B. Pismenny, I. Lesokhin, L. Liss, and H. Eran. TLS offload to network devices. In *Netdev 1.2*, 2016.

[86] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59(11):114–122, Oct. 2016.

[87] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040. Oct. 2007. URL: https://rfc-editor.org/rfc/rfc5040.txt.

[88] Y. Ren. High performance cloud with hardware acceleration. APNet 2018 (Accessed: Sep. 2018). 2018. URL: https://conferences.sigcomm.org/events/apnet2018/slides/yong.pdf.

[89] D. Riddoch and S. Pope. FPGA augmented ASICs: the time has come. In *HCS*, Aug. 2012, pp. 1–44.

[90] T. Rinta-aho, M. Karlstedt, and M. P. Desai. The Click2NetFPGA toolchain. In *USENIX ATC 2012*. USENIX, 2012, pp. 77–88.

[91] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP '11*. ACM, 2011, pp. 233–248.

[92] SAMSUNG. Samsung ARTIK cloud developer – CoAP. (Accessed: Sep. 2018). 2018. URL: https://developer.artik.cloud/documentation/data-management/coap.html.

[93] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *HotNets-XVI*. ACM, 2017, pp. 150–156.

[94] Selectel. FPGA-accelerators go into the clouds [russian]. (Accessed: Jan. 2019). 2018. URL: https://blog.selectel.ru/fpga-uskoriteli-uxodyat-v-oblaka/.

[95] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, June 1996.

[96] D. Sidler, Z. István, and G. Alonso. Low-latency TCP/IP stack for data center applications. In *FPL 2016*, Aug. 2016, pp. 1–4.

[97] Solarflare Communications, Inc. Application nanosecond TCP send (ANTS): from request to response in less than 250ns. (Accessed: Jan. 2019). 2015. URL: https://www.solarflare.com/Media/Default/PDFs/SF-114903-CD-LATEST-Solarflare_Application_Nanosecond_TCP_Send_Paper.pdf.

[98] S. Stanley. Ubiquitous SDN acceleration is coming. (Accessed: Jan. 2019). 2017. URL: https://www.lightreading.com/carrier-sdn/ubiquitous-sdn-acceleration-is-coming/a/d-id/738209.

[99] J. Strömbergson. Secworks/sha256: hardware implementation of the SHA-256 cryptographic hash function. (Accessed: Jan. 2019). 2018. URL: https://github.com/secworks/sha256.

[100] H. Stubbe. P4 compiler & interpreter: a survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, 47, 2017.

[101] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: a coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan. 2015.

[102] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: rapid prototyping of networking services. In *USENIX ATC 2017*. USENIX Association, 2017, pp. 459–471.

[103] S. Tanaka and C. Kozyrakis. High performance hardware-accelerated flash key-value store. In *NVMW 2014*, 2014.

[104] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow. Galapagos: a full stack approach to FPGA integration in the cloud. *IEEE Micro*, 38(6):18–24, Nov. 2018.

[105] The RoCE Initiative. RoCE introduction. (Accessed: Jan. 2019). 2016. URL: http://www.roceinitiative.org/roce-introduction/.

[106] Y. Tokusashi and H. Matsutani. A multilevel NOSQL cache design combining in-NIC and in-kernel caches. In *HOTI 2016*, Aug. 2016, pp. 60–67.

[107] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: the power of in-network computing. In *ReConFig'18*, Dec. 2018, pp. 1–8.

[108] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *EuroSys '19*. ACM, 2019, 21:1–21:16.

[109] D. Tong and V. Prasanna. High throughput sketch based online heavy hitter detection on FPGA. *SIGARCH Comput. Archit. News*, 43(4):70–75, Apr. 2016.

[110] A. Trivedi. Remote Direct Memory Access (RDMA) 101 – quick history lesson and introduction. (Accessed: Sep. 2018). 2011. URL: http://0x8086.blogspot.com/2011/11/remote-direct-memory-access-rdma-101.html.

[111] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: a rapid prototyping framework for P4. In *SOSR 2017*. ACM, 2017, pp. 122–135.

[112] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Quality of service support for fine-grained sharing on GPUs. In *ISCA '17*. ACM, 2017, pp. 269–281.

[113] P. Willmann, H.-y. Kim, S. Rixner, and V. S. Pai. An efficient programmable 10 gigabit Ethernet network interface card. In *HPCA-11*, Feb. 2005, pp. 96–107.

[114] Xilinx Inc. Vivado high-level synthesis. (Accessed: Jan. 2019). 2018. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[115] W. Xu. Hardware acceleration over NFV in China Mobile. OPNFV Plugfest. (Accessed: Jan. 2019). June 2018. URL: https://wiki.opnfv.org/download/attachments/20745096/opnfv_Acc.pdf.

[116] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers. Pagoda: fine-grained GPU resource virtualization for narrow tasks. In *PPoPP '17*. ACM, 2017, pp. 221–234.

[117] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang. G-NET: effective GPU sharing in NFV systems. In *NSDI '18*. USENIX Association, 2018, pp. 187–200.

[118] Q. Zhao, M. Iida, and T. Sueyoshi. A study of FPGA virtualization and accelerator scheduling. In *ETCD'17*. ACM, 2017, 3:1–3:4.

[119] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept. 2014.

E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers

Ming Liu
University of Washington

Simon Peter
The University of Texas at Austin

Arvind Krishnamurthy
University of Washington

Phitchaya Mangpo Phothilimthana*
University of California, Berkeley

Abstract

We investigate the use of SmartNIC-accelerated servers to execute microservice-based applications in the data center. By offloading suitable microservices to the SmartNIC's low-power processor, we can improve server energy-efficiency without latency loss. However, as a heterogeneous computing substrate in the data path of the host, SmartNICs bring several challenges to a microservice platform: network traffic routing and load balancing, microservice placement on heterogeneous hardware, and contention on shared SmartNIC resources.

We present E3, a microservice execution platform for SmartNIC-accelerated servers. E3 follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components to a SmartNIC to address the above-mentioned challenges. E3 employs three key techniques: ECMP-based load balancing via SmartNICs to the host, network topology-aware microservice placement, and a data-plane orchestrator that can detect SmartNIC overload. Our E3 prototype using Cavium LiquidIO SmartNICs shows that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices, including real-time analytics, an IoT hub, and virtual network functions.

1 Introduction

Energy-efficiency has become a major factor in data center design [80]. U.S. data centers consume an estimated 70 billion kilowatt-hours of energy per year (about 2% of total U.S. energy consumption) and as much as 57% of this energy is used by servers [22, 74]. Improving server energy-efficiency is thus imperative [17]. A recent option is the integration of low-power processors in server network interface cards (NICs). Examples are the Netronome Agilio-CX [59], Mellanox BlueField [51], Broadcom Stingray [13], and Cavium LiquidIO [15], which rely on ARM/MIPS-based processors and on-board memory. These SmartNICs can process microsecond-scale client requests but consume much less energy than server CPUs. By sharing idle power and the chassis with host servers, SmartNICs also promise to be more energy and cost efficient than other heterogeneous or low-power clusters. However, SmartNICs are not powerful enough to run large, monolithic cloud applications, preventing their offload.

Today, cloud applications are increasingly built as microservices, prompting us to revisit SmartNIC offload in the cloud. A microservice-based workload comprises loosely coupled processes, whose interaction is described via a dataflow graph. Microservices often have a small enough memory footprint for SmartNIC offload and their programming model efficiently supports transparent execution on heterogeneous platforms. Microservices are deployed via a microservice platform [3–5, 40] on shared datacenter infrastructure. These platforms abstract and allocate physical datacenter computing nodes, provide a reliable and available execution environment, and interact with deployed microservices through a set of common runtime APIs. Large-scale web services already use microservices on hundreds of thousands of servers [40, 41].

In this paper, we investigate efficient microservice execution on SmartNIC-accelerated servers. Specifically, we are exploring how to integrate multiple SmartNICs per server into a microservice platform with the goal of achieving better energy efficiency at minimum latency cost. However, transparently integrating SmartNICs into microservice platforms is non-trivial. Unlike traditional heterogeneous clusters, SmartNICs are collocated with their host servers, raising a number of issues. First, SmartNICs and hosts share the same MAC address. We require an efficient mechanism to route and load-balance traffic to hosts and SmartNICs. Second, SmartNICs sit in the host's data path and microservices running on a SmartNIC can interfere with microservices on the host. Microservices need to be appropriately placed to balance network-to-compute bandwidth. Finally, microservices can contend on shared SmartNIC resources, causing overload. We need to efficiently detect and prevent such situations.

We present E3, a microservice execution platform for SmartNIC-accelerated servers that addresses these issues. E3

*The author is now at Google.

follows the design philosophies of the Azure Service Fabric microservice platform [40] and extends key system components to allow transparent offload of microservices to a SmartNIC. To balance network request traffic among SmartNICs and the host, E3 employs equal-cost multipath (ECMP) load balancing at the top-of-rack (ToR) switch and provides high-performance PCIe communication mechanisms between host and SmartNICs. To balance computation demands, we introduce *HCM*, a hierarchical, communication-aware microservice placement algorithm, combined with a data-plane orchestrator that can detect and eliminate SmartNIC overload via microservice migration. This allows E3 to optimize server energy efficiency with minimal impact on client request latency.

We make the following contributions:

- We show why SmartNICs can improve energy efficiency over other forms of heterogeneous computation and how they should be integrated with data center servers and microservice platforms to provide efficient and transparent microservice execution (§2).
- We present the design of E3 (§3), a microservice runtime on SmartNIC-accelerated server systems. We present its implementation within a cluster of Xeon-based servers with up to 4 Cavium LiquidIO-based SmartNICs per server (§4).
- We evaluate energy and cost-efficiency, as well as client-observed request latency and throughput for common microservices, such as a real-time analytics framework, an IoT hub, and various virtual network functions, across various homogeneous and heterogeneous cluster configurations (§5). Our results show that offload of microservices to multiple SmartNICs per server with E3 improves cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% client-observed latency cost versus all other cluster configurations.

2 Background

Microservices simplify distributed application development and are a good match for low-power SmartNIC offload. Together, they are a promising avenue for improving server energy efficiency. We discuss this rationale, quantify the potential benefits, and outline the challenges of microservice offload to SmartNICs in this section.

2.1 Microservices

Microservices have become a critical component of today's data center infrastructure with a considerable and diverse workload footprint. Microsoft reports running microservices 24/7 on over 160K machines across the globe, including Azure SQL DB, Skype, Cortana, and IoT suite [40]. Google reports that Google Search, Ads, Gmail, video processing, flight search, and more, are deployed as microservices [41]. These microservices include large and small data and code footprints, long and short running times, billed by run-time

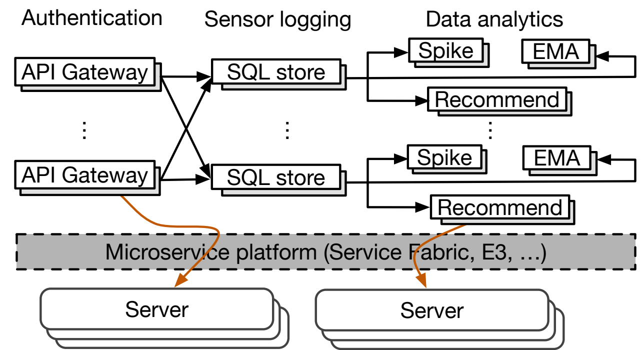


Figure 1: Thermostat analytics as DAG of microservices. The platform maps each DAG node to a physical computing node.

and by remote procedure call (RPC) [28]. What unifies these services is their software engineering philosophy.

Microservices use a modular design pattern, which simplifies distributed application design and deployment. Microservices are loosely-coupled, communicating through a set of common APIs, invoked via RPCs [86], and maintain state via *reliable collections* [40]. As a result, developers can take advantage of languages and libraries of their choice, while not having to worry about microservice placement, communication mechanisms, fault tolerance, or availability.

Microservices are also attractive to datacenter operators as they provide a way to improve server utilization. Microservices execute as light-weight processes that are easier to scale and migrate compared with a monolithic development approach. They can be activated upon incoming client requests, execute to request completion, and then swapped out.

A microservice platform, such as Azure Service Fabric [40], Amazon Lambda [3], Google Application Engine [4], or Nirmata [5], is a distributed system manager that enables isolated microservice execution on shared datacenter infrastructure. To do so, microservice platforms include the following components (cf. [40]): 1. *federation subsystem*, abstracting and grouping servers into a unified cluster that holds deployed applications; 2. *resource manager*, allocating computation resources to individual microservices based on their execution requirements; 3. *orchestrator*, dynamically scheduling and migrating microservices within the cluster based on node health information, microservice execution statistics, and service-level agreements (SLAs); 4. *transport subsystem*, providing (secure) point-to-point communication among various microservices; 5. *failover manager*, guaranteeing high availability/reliability through replication; 6. troubleshooting utilities, which assist developers with performance profiling/debugging and understanding microservice co-execution interference.

A microservice platform usually provides a number of programming models [10] that developers adhere to, like dataflow and actor-based. The models capture the execution requirements and describe the communication relationship among microservices. For example, the data-flow model (e.g. Amazon Datapipe [6], Google Cloudflow [29], Azure Data

Factory [55]) requires programmers to assemble microservices into a directed acyclic graph (DAG): nodes contain microservices that are interconnected via flow-controlled, lossless dataflow channels. These models bring attractive benefits for a heterogeneous platform since they explicitly express concurrency and communication, enabling the platform to transparently map it to the available hardware [68, 70]. Figure 1 shows an IoT thermostat analytics application [54] consisting of microservices arranged in 3 stages: 1. Thermostat sensor updates are authenticated by the API gateway; 2. Updates are logged into a SQL store sharded by a thermostat identifier; 3. SQL store updates trigger data analytic tasks (e.g, spike detection, moving average, and recommendation) based on thresholds. The dataflow programming model allows the SQL store sharding factor to be dynamically adjusted to scale the application with the number of thermostats reporting. Reliable collections ensure state consistency when re-sharding and the microservice platform automatically migrates and deploys DAG nodes to available hardware resources.

A microservice can be stateful or stateless. Stateless microservices have no persistent storage and only keep state within request context. They are easy to scale, migrate, and replicate, and they usually rely on other microservices for stateful tasks (e.g., a database engine). Stateful microservices use platform APIs to access durable state, allowing the platform full control over data placement. For example, Service Fabric provides reliable collections [40], a collection of data structures that automatically persist mutations. Durable storage is typically disaggregated for microservices and accessed over the network. The use of platform APIs to maintain state allows for fast service migration compared with traditional virtual machine migration [19], as the stateful working set is directly observed by the platform. All microservices in Figure 1 are stateful. We describe further microservices in §4.

2.2 SmartNICs

SmartNICs have appeared on the market [15, 51, 59] and in the datacenter [25]. SmartNICs include computing units, memory, traffic managers, DMA engines, TX/RX ports, and several hardware accelerators for packet processing, such as cryptography and pattern matching engines. Unlike traditional accelerators, SmartNICs integrate the accelerator with the NIC. This allows them to process network requests in-line, at much lower latency than other types of accelerators.

Two kinds of SmartNIC exist: (1) *general-purpose*, which allows transparent microservice offload and is the architecture we consider. For example, Mellanox BlueField [51] has 16 ARMv8 A72 cores with 2×100GE ports and Cavium LiquidIO [15] has 12 cnMIPS cores with 2×10GE ports. These SmartNICs are able to run full operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC's IO engines. (2) FPGA and ASIC based SmartNICs target highly specialized applications. Examples include match-and-action processing [25, 43] for network dataplanes, NPUs [26], and TPUs [39] for deep neural network inference acceleration. FPGAs and ASICs do not support transparent microservice offload. However, they can be combined with general-purpose SmartNICs.

A SmartNIC-accelerated server is a commodity server with one or more SmartNICs. Host and SmartNIC processors do not share thermal, memory, or cache coherence domains, and communicate via DMA engines over PCIe. This allows them to operate as independent, heterogeneous computers, while sharing a power domain and its idle power.

SmartNICs hold promise for improving server energy-efficiency when compared to other heterogeneous computing approaches. For example, racks populated with low-power servers [8] or a heterogeneous mix of servers, suffer from high idle energy draw, as each server requires energy to power its chassis, including fans and devices, and its own ToR switch port. System-on-chip designs with asymmetric performance, such as ARM's big.LITTLE [38] and DynamIQ [2] architectures, and AMD's heterogeneous system architecture (HSA) [7], which combines a GPU with a CPU on the same die, have scalability limits due to the shared thermal design point (TDP). These architectures presently scale to a maximum of 8 cores, making them more applicable to mobile than to server applications. GPGPUs and single instruction multiple threads (SIMT) architectures, such as Intel's Xeon Phi [36] and HP Moonshot [34], are optimized for computational throughput and the extra interconnect hop prevents these accelerators from running latency-sensitive microservices efficiently [57]. SmartNICs are not encumbered by these problems and can thus be used to balance the power draw of latency-sensitive services efficiently.

2.3 Benefits of SmartNIC Offload

We quantify the potential benefit of using SmartNICs for microservices on energy efficiency and request latency. To do so, we choose two identical commodity servers and equip one with a traditional 10GbE Intel X710 NIC and the other with a 10GbE Cavium LiquidIO SmartNIC. Then we evaluate 16 different microservices (detailed in §4) on these two servers with synthetic benchmarks of random 512B requests. We measure request throughput, wall power consumed at peak throughput (defined as the knee of the latency-throughput graph, where queueing delay is minimal) and when idle, as well as client-observed, average/tail request latency in a closed loop. We use host cores on the traditional server and SmartNIC cores on the SmartNIC server for microservice execution. We use as many identical microservice instances, CPUs, and client machines as necessary to attain peak throughput and put unused CPUs to their deepest sleep state. The SmartNIC does not support per-core low power states and always keeps all 12 cores active, diminishing SmartNIC energy efficiency results somewhat. The SmartNIC microservice runtime system uses a kernel-

Microservice	Host (Linux)						Host (DPDK)							SmartNIC					
	RPS	W	C	L	99%	RPJ	RPS	W	C	L	99%	RPJ	%	RPS	W	L	99%	RPJ	×
IPsec	821.3K	117.0	12	1.8	6.6	7.0K	911.9K	112.1	12	1.7	5.2	8.1K	15.9	1851.1K	23.4	0.2	0.8	79.0K	9.7
BM25	91.9K	116.4	12	40.3	205.8	0.8K	99.5K	110.0	12	30.7	155.6	0.9K	14.5	394.1K	19.2	4.1	12.4	20.6K	22.8
NIDS	1781.1K	111.0	12	0.06	0.2	16.1K	1841.1K	106.8	12	0.05	0.15	17.2K	7.4	1988.8K	23.4	0.03	0.1	84.8K	4.9
Recommend	3.6K	109.4	12	86.6	477.0	0.03K	4.1K	111.7	12	78.7	358.6	0.04K	11.6	12.8K	18.9	21.3	123.6	0.7K	18.4
NATv4	1889.6K	72.1	8	0.04	0.1	26.2K	1917.5K	52.1	4	0.04	0.1	36.8K	40.4	2053.1K	23.6	0.03	0.09	86.9K	2.4
Count	1960.8K	68.1	6	0.07	0.1	28.8K	1960.0K	48.6	4	0.03	0.1	40.3K	40.0	2016.8K	21.0	0.03	0.09	96.1K	2.4
EMA	1966.1K	72.7	8	0.04	0.2	27.0K	2009.2K	52.1	4	0.03	0.09	38.6K	42.8	2052.0K	22.0	0.03	0.08	93.5K	2.4
KVS	1946.2K	48.6	8	0.04	0.1	40.0K	2005.0K	33.6	2	0.04	0.1	59.6K	49.0	2033.4K	21.6	0.03	0.1	97.1K	1.6
Flow mon.	1944.1K	70.9	8	0.04	0.1	27.4K	2014.4K	49.8	4	0.03	0.09	40.4K	47.4	2032.6K	24.3	0.03	0.08	83.6K	2.1
DDoS	1989.5K	111.2	12	0.05	0.2	17.9K	1844.8K	105.7	12	0.05	0.2	17.4K	**-3.0**	1952.5K	24.3	0.03	0.1	80.4K	4.6
KNN	42.2K	118.3	12	53.7	163.4	0.4K	42.4K	110.4	12	45.8	161.3	0.4K	7.5	29.9K	20.0	20.6	80.3	1.5K	3.9
Spike	91.9K	112.5	12	29.3	94.5	0.8K	104.3K	112.3	12	25.7	83.0	0.9K	13.7	73.8K	23.5	9.0	50.3	3.1K	3.4
Bayes	12.1K	113.9	12	82.0	406.5	0.1K	13.7K	112.0	12	80.6	400.5	0.1K	14.8	1.6K	19.5	41.9	164.7	0.08K	**0.7**
API gw	1537.6K	108.5	12	0.9	3.2	14.2K	1584.3K	110.6	12	0.8	2.7	14.3K	1.1	124.5K	24.7	8.5	403.6	5.0K	**0.4**
Top ranker	711.9K	119.7	12	4.0	15.0	5.9K	771.9K	109.2	12	3.5	12.3	7.1K	18.9	14.8K	20.3	31.1	154.9	0.7K	**0.1**
SQL	463.3K	114.7	12	6.9	31.1	4.0K	528.0K	113.0	12	6.7	29.5	4.7K	15.7	39.5K	18.8	29.5	104.2	2.1K	**0.4**

Table 1: Microservice comparison among host (Linux and DPDK) and SmartNIC. RPS = Throughput (requests/s), W = Active power (W), C = Number of active cores, L = Average latency (ms), 99% = 99th percentile latency, RPJ = Energy efficiency (requests/Joule).

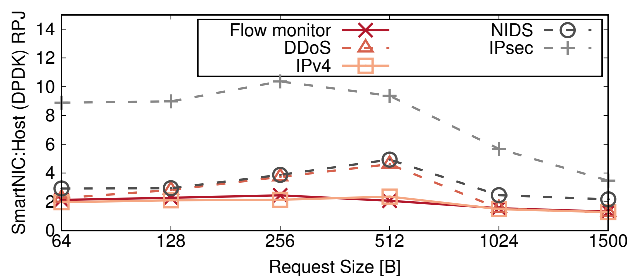


Figure 2: Request size impact on SmartNIC RPJ benefits.

bypass network stack (cf. §4). To break out kernel overheads from the host experiments, we run all microservices on the host in two configurations: 1. Linux kernel network stack; 2. kernel-bypass network stack [63], based on Intel's DPDK [1].

Table 1 presents measured peak request throughput, active power (wall power at peak throughput minus idle wall power), number of active cores, (tail-)latency, and energy efficiency, averaged over 3 runs. Active power allows a direct comparison of host to SmartNIC processor power draw. Energy efficiency equals throughput divided by active power.

Kernel overhead. We first analyze the overhead of in-kernel networking on the host (Linux versus DPDK). As expected, the kernel-bypass networking stack performs better than the in-kernel one. On average, it improves energy efficiency by 21% (% column in Table 1) and reduces tail latency by 16%. Energy efficiency improves because (1) DPDK achieves similar throughput with fewer cores; (2) at peak server CPU utilization, DPDK delivers higher throughput.

SmartNIC performance. SmartNIC execution improves the energy efficiency of 12 of the measured microservices by a geometric mean of 6.5× compared with host execution using kernel bypass (× column in Table 1). The SmartNIC consumes at most 24.7W active power to execute these microservices while the host processor consumes up to 113W. IPSec, BM25, Recommend, and NIDS particularly benefit from various SmartNIC hardware accelerators (crypto coprocessor, fetch-and-add atomic units, floating point engines, and pattern matching units). NATv4, Count, EMA, KVS, Flow monitor, and DDoS can take advantage of the computational bandwidth and fast memory interconnect of the SmartNIC. In these cases, the energy efficiency comes not just from the lower power consumed by the SmartNIC, but also from peak throughput improvements versus the host processor. KNN and Spike attain lower throughput on the SmartNIC. However, since the SmartNIC consumes less power, the overall energy efficiency is still better than the host. For all of these microservices, the SmartNIC also improves client-observed latency. This is due to the hardware accelerated packet buffers and the elimination of PCIe bus traversals. SmartNICs can reduce average and tail latency by a geometric mean of 45.3% and 45.4% versus host execution, respectively.

The host outperforms the SmartNIC for Top ranker, Bayes classifier, SQL, and API gateway by a geometric mean of 4.1× in energy efficiency, 41.2% and 30.0% in average and tail latency reduction. These microservices are branch-heavy with large working sets that are not handled well by the simpler cache hierarchy of the SmartNIC. Moreover, the API gateway uses double floating point numbers for the rate limiter implementation, which the SmartNIC emulates in software.

Request size impact. SmartNIC performance depends also on request size. To demonstrate this, we vary the request size of our synthetic workload and evaluate SmartNIC energy efficiency benefits of 5 microservices versus host execution. Figure 2 shows that with small (≤128B) requests, SmartNIC benefit of IPSec, NIDS, and DDoS is smaller. Small requests are more computation intensive and we are limited by the SmartNIC's wimpy cores. SmartNIC offload hits a sweet-spot at 256–512B request size, where the benefit almost doubles. Here, network and compute bandwidth utilization are balanced for the SmartNIC. At larger request sizes, we are network bandwidth limited, allowing us to put host CPUs to sleep and SmartNIC benefits again diminish. This can be seen in particular for IPsec, which outperforms on the SmartNIC due

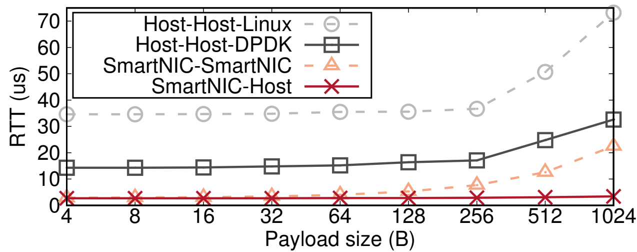


Figure 3: Average RTT (3 runs) of different communication mechanisms in a SmartNIC-accelerated server.

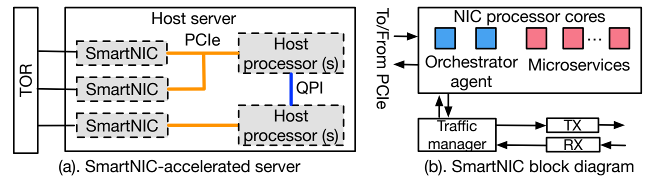


Figure 4: Hardware and software architecture of E3.

to hardware cryptography acceleration, but still diminishes with larger request sizes. We conclude that request size has a major impact on the benefit of SmartNIC offload. Measuring it is necessary to make good offload choices.

We conclude that SmartNIC offload can provide large energy efficiency and latency benefits for many microservices. However, it is not a panacea. Computation and memory intensive microservices are more suitable to run on the host processor. We need an efficient method to define and monitor critical SmartNIC offload criteria for microservices.

2.4 Challenges of SmartNIC Offload

While there are quantifiable benefits, offloading microservices to SmartNICs brings a number of additional challenges:

- SmartNICs share the same Ethernet MAC address with the host server. Layer 2 switching is not enough to route traffic between SmartNICs and host servers. We require a different switching scheme that can balance traffic and provide fault tolerance when a server equips multiple SmartNICs.
- Microservice platforms assume uniform communication performance among all computing nodes. However, Figure 3 shows that SmartNIC-Host (via PCIe) and SmartNIC-SmartNIC (via ToR switch) communication round-trip-time (RTT) is up to 83.3% and 86.2% lower than host-host (via ToR switch) kernel-bypass communication. We have to consider this topology effect to achieve good performance.
- Microservices share SmartNIC resources and contend with SmartNIC firmware for cache and memory bandwidth. This can create a head-of-line blocking problem for network packet exchange with both SmartNIC and host. Prolonged head-of-line blocking can result in denial of service to unrelated microservices and is more severe than transient sources of interference, such as network congestion. We need to sufficiently isolate SmartNIC-offloaded microservices from firmware to guarantee quality of service.

3 E3 Microservice Platform

We present the E3 microservice platform for SmartNIC-accelerated servers. Our goal is to maximize microservice energy efficiency at scale. Energy efficiency is the ratio of microservice throughput and cluster power draw. Power draw is determined by our choice of SmartNIC-acceleration, while E3 focuses on maximizing microservice throughput on this heterogeneous architecture. We describe how we support microservice offload to a SmartNIC and address the request routing, microservice placement, and scheduling challenges.

E3 overview. E3 is a distributed microservice execution platform. We follow the design philosophies of Azure Service Fabric [40] but add energy efficiency as a design requirement. Figure 4 shows the hardware and software architecture of E3. E3 runs in a typical datacenter, where servers are grouped into racks, with a ToR switch per rack. Each server is equipped with one or more SmartNICs, and each SmartNIC is connected to the ToR. This creates a new topology where host processors are reachable via any of the SmartNICs (Figure 4-a). SmartNICs within the same server also have multiple communication options—via the ToR or PCIe (§3.1).

Programming model. E3 uses a dataflow programming model. Programmers assemble microservices into a DAG of microservice nodes interconnected via channels in the direction of RPC flow (cf. Figure 1). A channel provides lossless data communication between nodes. A DAG in E3 describes all RPC and execution paths of a single microservice application, but multiple DAGs may coexist and execute concurrently. E3 is responsible for mapping DAGs to computational nodes.

Software stack. E3 employs a central, replicated cluster resource controller [40] and a microservice runtime on each host and SmartNIC. The resource controller includes four components: (1) traffic control, responsible for routing and load balancing requests between different microservices; (2) control-plane manager, placing microservice instances on cluster nodes; (3) data-plane orchestrator, dynamically migrates microservices across cluster nodes; (4) failover/replication manager, providing failover and node membership management using consistent hashing [75]. The microservice runtime includes an execution engine, an orchestrator agent, and a communication subsystem, described next.

Execution engine. E3 executes each microservice as a multi-threaded process, either on the SmartNIC or on the host. The host runs Linux. The SmartNIC runs a lightweight firmware. Microservices interact only via microservice APIs,

allowing E3 to abstract from the OS. SmartNIC and host support hardware virtual memory for microservice confinement. E3 is work-conserving and runs requests to completion. It leverages a round-robin policy for steering incoming requests to cores, context switching cores if needed.

Orchestrator agent. Each node runs an orchestrator agent to periodically monitor and report runtime execution characteristics to the resource controller. The information is used by (1) the failover manager to determine cluster health and (2) the data-plane orchestrator to monitor the execution performance of each microservice and make migration decisions. On the host, the agent runs as a separate process. On the SmartNIC, the agent runs on dedicated cores (blue in Figure 4-b) and a traffic manager hardware block exchanges packets between the NIC MAC ports and the agent. For each packet, the agent determines the destination (network, host, or SmartNIC core).

3.1 Communication Subsystem

E3 leverages various communication mechanisms, depending on where communicating microservices are located.

Remote communication. When communicating among host cores across different servers, E3 uses the Linux network stack. SmartNIC remote communication uses a user-level network stack [63].

Local SmartNIC-host communication. SmartNIC and host cores on the same server communicate via PCIe. Prior work has extensively explored communication channels via PCIe [47, 48, 60], and we adopt their design. High-throughput messaging for PCIe interconnects requires leveraging multiple DMA engines in parallel. E3 takes advantage of the eight DMA engines on the LiquidIO, which can concurrently issue scatter/gather requests.

Local SmartNIC-SmartNIC communication. SmartNICs in the same host can use three methods for communication. 1. Using the host to relay requests, involving two data transfers over PCIe and pointer manipulation on the host, increasing latency. 2. PCIe peer-to-peer [23], which is supported on most SmartNICs [15, 51, 59]. However, the bandwidth of peer-to-peer PCIe communication is capped in a NUMA system when the communication passes between sockets [57]. 3. ToR switch. We take the third approach and our experiments show that this approach incurs lower latency and achieves higher bandwidth than the first two.

3.2 Addressing and Routing

Since SmartNICs and their host servers share Ethernet MAC addresses, we have to use an addressing/routing scheme to distinguish between these entities and load balance across them. For illustration, assume we have a server with two SmartNICs; each NIC has one MAC port. If remote microservices communicate with this server, there will be two possible paths and each might be congested.

We use equal-cost multi-path (ECMP) [84] routing on the ToR switch to route and balance load among these ports. We assign each SmartNIC and the host its own IP. We then configure the ToR switch to route to SmartNICs directly via the attached ToR switch port and an ECMP route to the host IP via any of the ports. The E3 communication subsystem on each SmartNIC differentiates by destination IP address whether an incoming packet is for the SmartNIC or the host. On the host, we take advantage of NIC teaming [56] (also know as port trunking) to bond all related SmartNIC ports into a single logical interface, and then apply the dynamic link aggregation policy (supporting IEEE 802.3ad protocol). ECMP automatically balances connections to the host over all available ports. If a link or SmartNIC fails, ECMP will automatically rebalance new connections via the remaining links, improving host availability.

3.3 Control-plane Manager

The control-plane manager is responsible for energy-efficient microservice placement. This is a computing intensive operation due to the large search space with myriad constraints. Hence, it is done on the control plane. Service Fabric uses simulated annealing, a well-known approximate algorithm, to solve microservice placement. It considers three types of constraints: (1) currently available resources of each computing node (memory, disk, CPU, network bandwidth); (2) computing node runtime statistics (aggregate outstanding microservice requests); (3) individual microservice execution behavior (average request size, request execution time and frequency, diurnal variation, etc.). Service Fabric ignores network topology and favors spreading load over multiple nodes.

E3 extends this algorithm to support bump-in-the-wire SmartNICs, considering network topology. We categorize *computing nodes* (host or SmartNIC processors) into different levels of communication distance and perform a search from the closest to the furthest. We present the HCM algorithm (Algorithm 1). HCM takes as input the microservice DAG G and source nodes V_{src}, as well as the cluster topology T, including runtime statistics for each computing node (as collected). HCM performs a breadth-first traversal of G to map microservices to cluster computing nodes (*MS_DAG_TRAVERSE*).

If not already deployed (*get_deployed_node*), HCM (via *MS_DAG_TRAVERSE*) assigns a microservice V to a computing node N via the *find_first_fit* function (lines 9-11) and deploys it via *set_deployed_node*. *find_first_fit* is a greedy algorithm that returns the first computing node that satisfies the microservice constraints (via its resource and runtime statistics) without considering communication cost. If no such node is found, it returns a node closest to the constraints. Next,

Algorithm 1 HCM microservice placement algorithm

```
1:  G : microservice DAG graph
2:  V_src : source microservice node(s) of the DAG
3:  T : server cluster topology graph
4:  procedure MS_DAG_TRAVERSE(G, V_src, T)
5:      Q.enqueue(V_src)                                ▷ Let Q be a queue
6:      while Q is not empty do
7:          V ← Q.dequeue()
8:          N ← get_deployed_node(V)
9:          if N is NULL then
10:             N ← find_first_fit(V, T)
11:             set_deployed_node(V, N)
12:         for W in all direct descendants of V in G do
13:             N_W ← MS_PLACE(W, N, T)
14:             set_deployed_node(W, N_W)
15:             Q.enqueue(W)
16:
17: V : microservice to place
18: N : computational node of V's ancestor
19: T : server cluster topology graph
20: procedure MS_PLACE(V, N, T)
21:     Topo ← get_hierarchical_topo(N, T)
22:     for L in all Topo.Levels do
23:         N ← find_best_fit(V, Topo.node_list(L))
24:         if N is not NULL then
25:             return N
26:     return find_first_fit(V, T)                     ▷ Ignore topology
```

for the descendant microservices of a node V (lines 12-15), HCM assigns them to computing nodes based on their communication distance to V (*MS_PLACE*). To do so, HCM first computes the *hierarchical topology representation* of computing node N via *get_hierarchical_topo*. Each level in the hierarchical topology includes computing nodes that require a similar communication mechanism, starting with the closest. For example, in a single rack there are four levels in this order: 1. The same computing node as V; 2. An adjacent computing node on the same server; 3. A SmartNIC computing node on an adjacent server; 4. A host computing node on an adjacent server. If there are multiple nodes in the same level, HCM uses *find_best_fit* to find the best fit, according to resource constraints. If no node in the hierarchical topology fits the constraints, we fall back to *find_first_fit*.

3.4 Data-plane Orchestrator

The data-plane orchestrator is responsible for detecting load changes and migrating microservices in response to these changes among computational nodes at run-time. To do so, we piggypack several measurements onto the periodic node health reports made by orchestrator agents to the resource controller: This approach is lightweight and integrates well with runtime execution. We believe that our proposed methods can also be used in other microservice schedulers [33, 62, 66].

In this section, we introduce the additional techniques implemented in our data-plane orchestrator to mitigate issues of SmartNIC overload caused by compute-intensive microservices. These can interfere with the SmartNIC's traffic manager, starving the host of network packets. They can also simply execute too slowly on the SmartNIC to be able to catch up with the incoming request rate.

Host starvation. This issue is caused by head-of-line blocking of network traffic due to microservice interference with firmware on SmartNIC memory/cache. It is typically caused by a single compute-intensive microservice overloading the SmartNIC. To alleviate this problem, we monitor the incoming/outgoing network throughput and packet queue depth at the traffic manager. If network bandwidth is under-utilized, but there is a standing queue at the traffic manager, the SmartNIC is overloaded, and we need to migrate microservices.

Microservice overload. This issue is caused by microservices in aggregate requiring more computational bandwidth than the SmartNIC can offer, typically because too many microservices are placed on the same SmartNIC. To detect this problem, we periodically monitor the execution time of each microservice and compare to its exponential moving average. When the difference is negative and larger than 20%, we assume a microservice overload and trigger microservice migration. The threshold was determined empirically.

Microservice migration. For either issue, the orchestrator will migrate the microservice with the highest CPU utilization to the host. To do so, it uses a cold migration approach, similar to other microservice platforms. Specifically, when the orchestrator makes a migration decision, it will first push the microservice binary to the new destination, and then notify the runtime of the old node to (1) remove the microservice instance from the execution engine; (2) clean up and free any local resources; (3) migrate the working state, as represented by reliable collections [40], to the destination. After the orchestrator receives a confirmation from the original node, it will update connections and restart the microservice execution on the new node.

3.5 Failover/Replication Manager

Since SmartNICs share the same power supply as their host server, our failover manager treats all SmartNICs and the host to be in the same fault domain [40], avoiding replica placement within the same. Replication for fault tolerance is typically done across different racks of the same datacenter or across datacenters, and there is no impact from placing SmartNICs in the same failure domain as hosts.

Microservice	S	Description
IPsec		Authenticates (SHA-1) & encrypts (AES-CBC-128) NATv4 [42]
BM25		Search engine ranking function [85], e.g., ElasticSearch
NATv4		IPv4 network address translation using DIR-24-8-BASIC [32]
NIDS		Network intrusion detection w/ aho-corasick parallel match [81]
Count	✓	Item frequency counting based on a bitmap [42]
EMA	✓	Exponential moving average (EMA) for data streams [83]
KVS	✓	Hashtable-based in-memory key-value store [24]
Flow mon.	✓	Flow monitoring system using count-min sketch [42]
DDoS	✓	Entropy-based DDoS detection [58]
Recommend	✓	Recommendation system using collaborative filtering [82]
KNN		Classifier using the K-nearest neighbours algorithm [87]
Spike	✓	Spike detector from a data stream using Z-score [72]
Bayes		Naive Bayes classifier based on *maximum a posteriori* [49]
API gw	✓	API rate limiter and authentication gateway [9]
Top Ranker	✓	Top-K ranker using quicksort [78]
SQL	✓	In-memory SQL database [52]

Table 2: 16 microservices implemented on E3. S = Stateful.

Application	Description	N	Microservices
NFV-FIN	Flow monitoring [42, 64]	72	Flow mon., IPsec, NIDS
NFV-DIN	Intrusion detection [64, 88]	60	DDoS, NATv4, NIDS
NFV-IFID	IPsec gateway [42, 88]	84	NATv4, Flow mon., IPsec, DDoS
RTA-PTC	Twitter analytics [78]	60	Count, Top Ranker, KNN
RTA-SF	Spam filter [35]	96	Spike, Count, KVS, Bayes
RTA-SHM	Server health mon. [37]	84	Count, EMA, SQL, BM25
IOT-DH	IoT data hub [77]	108	API gw, Count, KNN, KVS, SQL
IOT-TS	Thermostat [54]	108	API,EMA,Spike,Recommend,SQL

Table 3: 8 microservice applications. N = # of DAG nodes.

4 Implementation

Host software stack. The E3 resource controller and host runtime are implemented in 1,287 and 3,617 lines of C (LOC), respectively, on Ubuntu 16.04. Communication among co-located microservices uses per-core, multi-producer, single-consumer FIFO queues in shared memory. Our prototype uses UDP for all network communication.

SmartNIC runtime. The E3 SmartNIC runtime is built in 3,885 LOC on top of the Cavium CDK [16], with a user-level network stack. Each microservice runs on a set of non-preemptive hardware threads. Our implementation takes advantage of a number of hardware accelerator libraries. We use (1) a hardware managed memory manager to store the state of each microsevice, (2) the hardware traffic controller for Ethernet MAC packet management, and (3) atomic fetch-and-add units to gather performance statistics. We use page protection of the cnMIPS architecture to confine microservices.

Microservices. We implemented 16 popular microservices on E3, as shown in Table 2, in an aggregate 6,966 LOC. Six of the services are stateless or use read-only state that is modified only via the cluster control plane. The remaining services are stateful and use reliable collections to maintain their state. When running on the SmartNIC, IPsec and API gateway can use the crypto coprocessor (105 LOC), while Recommend and NIDS can take advantage of the deterministic finite automata unit (65 LOC). For Count, EMA, KVS, and Flow monitor, our compiler automatically uses the dedicated atomic fetch-and-add units on the SmartNIC. When performing single-precision floating-point computations (EMA, KNN, Spike, Bayes), our compiler generates FPU code on the SmartNIC. Double-precision floating-point calculations (API gateway) are software emulated. E3 reliable collections currently only support hashtables and arrays, preventing us from migrating the SQL engine. We thus constrain the control-plane manager to pin SQL instances to host processors.

System/Cluster	Cost [$]	BC	WC	Mem	Idle	Peak	Bw
Beefy	4,500	12	0	64	83	201	20
Wimpy	2,209	0	32	2	79	95	20
Type1-SmartNIC	4,650	12	12	68	98	222	20
Type2-SmartNIC	6,750	16	48	144	145	252	40
SuperBeefy	12,550	24	0	192	77	256	80
4×Beefy	18,000	48	0	256	332	804	80
4×Wimpy	8,836	0	128	8	316	380	80
2×B.+2×W.	13,018	24	64	132	324	592	80
2×Type2-SmartNIC	13,500	32	96	288	290	504	80
1×SuperBeefy	12,550	24	0	192	77	256	80

Table 4: Evaluated systems and clusters. BC = Beefy cores, WC = Wimpy cores, Mem = Memory (GB), Idle and Peak power (W), Bw = Network bandwidth (Gb/s).

Applications. Based on these microservices, we develop eight applications across three application domains: (1) Distributed real-time analytics (RTA), such as Apache Storm [78], implemented as a dataflow processing graph of workers that pass data tuples in real time to trigger computations; (2) Network function (NF) virtualization (NFV) [61], which is used to build cloud-scale network middleboxes, software switches, and enterprise IT networks, by chaining NFs; (3) An IoT hub (IOT) [53], which gathers sensor data from edge devices and generates events for further processing (e.g., spike detection, classifier) [?, 77]. To maximize throughput, applications may shard and replicate microservices, resulting in a DAG node count larger than the involved microservice types. Table 3 presents the microservice types involved in each application, the deployed DAG node count, and references the workloads used for evaluation. The workloads are trace-based and synthetic benchmarks, validated against realistic scenarios. The average and maximum node fanouts among our applications are 6 and 12, respectively. Figure1 shows IOT-TS as an example. IOT-TS is sharded into 6×API, 12×SQL, 12×EMA, 12×Spike, and 12×recommend and each microservice has one backup replica.

5 Evaluation

Our evaluation aims to answer the following questions:

1. What is the energy efficiency benefit of microservice SmartNIC-offload? Is it proportional to client load? What is the latency cost? (§5.1)
2. Does E3 overcome the challenges of SmartNIC-offload? (§5.2, §5.3, §5.4)
3. Do SmartNIC-accelerated servers provide better total cost of ownership than other cluster architectures? (§5.5)
4. How does E3 perform at scale? (§5.6)

Experimental setup. Our experiments run on a set of clusters (Table 4 presents the server and cluster configurations), attached to an Arista DCS-7050S ToR switch. Beefy is a Supermicro 1U server, with a 12-core E5-2680 v3 processor at 2.5GHz, and a dual-port 10Gbps Intel X710 NIC. Wimpy is ThunderX-like, with a CN6880 processor (32 cnMIPS64 cores running at 1.2GHz), and a dual-port 10Gbps NIC. SuperBeefy is a Supermicro 2U machine, with a 24-core Xeon Platinum 8160 CPU at 2.1GHz, and a dual-port 40Gbps Intel XL710 NIC. Our SmartNIC is the Cavium LiquidIOII [15], with one OCTEON processor with 12 cnMIPS64 cores at 1.2GHz, 4GB memory, and two 10Gbps ports. Based on this, we build two SmartNIC servers: Type1 is Beefy, but swaps the X710 10Gbps NIC with the Cavium LiquidIOII; Type2 is a 2U server with two 8-core Intel E5-2620 processors at 2.1GHz, 128GB memory, and 4 SmartNICs. All servers have a Seagate HDD. We build the clusters such that each has the same amount of aggregate network bandwidth. This allows us to compare energy efficiency based on the compute bandwidth of the clusters, without varying network bandwidth. We also exclude the switch from our cost and energy evaluations, as each cluster uses an identical number of switch ports.

We measure server power consumption using the servers' IPMI data center management interface (DCMI), cross-checked by a Watts Up wall power meter. Throughput and average/tail latency across 3 runs are measured from clients (Beefy machines), of which we provide as many as necessary. We enable hyper-threading and use the `Intel_pstate` governor for power management. All benchmarks in this section report energy efficiency as throughput over server/cluster **wall power** (not just active power).

5.1 Benefit and Cost of SmartNIC-Offload

Peak utilization. We evaluate the latency and energy efficiency of using SmartNICs for microservice applications, compared to homogeneous clusters. We compare 3×Beefy to 3×Type1-SmartNIC, to ensure that microservices also communicate remotely. We focus first on peak utilization, which is desirable for energy efficiency, as it amortizes idle power draw. To do so, we deploy as many instances of each application and apply as much client load as necessary to maximize request throughput without overloading the cluster, as determined by the knee of the latency-throughput curve.

Figure 5 shows that Type1-SmartNIC achieves an average 2.5×, 1.3×, and 1.3× better energy efficiency across the NFV, RTA, and IOT application classes, respectively. This goes along with 43.3%, 92.3%, and 80.4% average latency savings and 35.5%, 90.4%, 88.6% 99th percentile latency savings, respectively. NFV-FIN gains the most—3× better energy efficiency—because E3 is able to run all microservices on the SmartNICs. RTA-PTC benefits the least—12% energy efficiency improvement at 4% average and tail latency cost—as E3 only places the Count microservice on the SmartNIC and migrates the rest to the host.

Power proportionality. This experiment evaluates the power proportionality of E3 (energy efficiency at lower than peak utilization). Using 3×Type1-SmartNIC, we choose an application from each class (NFV-FIN, RTA-SHM, and IOT-TS) and vary the offered request load between idle and peak via a client side request limiter. Figure 8 shows that RTA-SHM and IOT-TS are power proportional. NFV-FIN is not power proportional but also draws negligible power. NFV-FIN runs all microservices on the SmartNICs, which have low active power, but the cnMIPS architecture has no per-core sleep states.

We conclude that applications can benefit from E3's microservice offload to SmartNICs, in particular at peak cluster utilization. Peak cluster utilization is desirable for energy efficiency and microservices make it more common due to light-weight migration. However, transient periods of low load can occur and E3 draws power proportional to request load. We can apply insights from Prekas, et al. [65] to reduce polling overheads and improve power proportionality further.

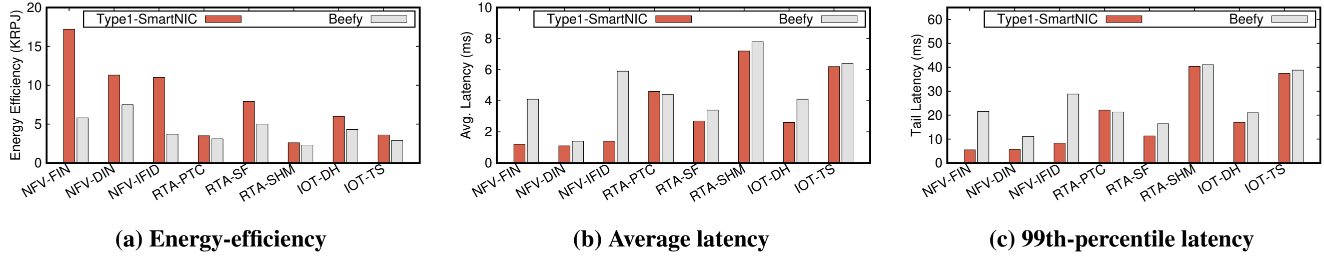
5.2 Avoiding Host Starvation

We show that E3's data-plane orchestrator prevents host starvation by identifying head-of-line blocking of network traffic. To do so, we use 3×Type1-SmartNIC and place as many microservices on the SmartNIC as fit in memory. E3 identifies the microservices that cause interference (Top Ranker in RTA-PTC, Spike in RTA-SF, API gateway in IOT-DH and IOT-TS) and migrates them to the host. As shown in Figure 7, our approach achieves up to 29× better energy efficiency and up to 89% latency reduction across RTA-PTC, RTS-SF, IOT-DH, and IOT-TS. For the other applications, our traffic engine has little effect because the initial microservice assignment already put the memory intensive microservices on the host.

5.3 Sharing SmartNIC and Host Bandwidth

This experiment evaluates the benefits of sharing SmartNIC network bandwidth with the host. We compare two Type2-SmartNIC configurations: 1. Sharing aggregate network bandwidth among host and SmartNICs, using ECMP to balance host traffic over SmartNIC ports; 2. Replacing one SmartNIC with an Intel X710 NIC used exclusively to route traffic to the host. To emphasize the load balancing benefits, we always place the client-facing microservices on the host server. Note that SmartNIC-offloaded microservices still exchange network traffic (when communicating remotely or among SmartNICs) and interfere with host traffic.

Figure 9 shows that load balancing improves application throughput up to 2.9× and cluster energy efficiency up to 2.7× (NFV-FIN). Available host network bandwidth when sharing SmartNICs can be up to 4× that of the dedicated



(a) Energy-efficiency **(b) Average latency** **(c) 99th-percentile latency**

Figure 5: Energy-efficiency, average/tail latency comparison between Type1-SmartNIC and Beefy at peak utilization.

Cluster	NFV-FIN	RTA-SHM	IOT-TS
4×Beefy	5.1	1.9	2.7
4×Wimpy	29.9	0.4	0.1
2×B.+2×W.	8.2	1.4	1.9
2×Type2-SmartNIC	29.0	4.5	6.1
1×SuperBeefy	8.8	2.9	5.0

Table 5: Energy efficiency across five clusters (KRPJ).

NIC, which balances better with the host compute bandwidth. With a dedicated NIC, host processors can starve for network bandwidth. IOT-TS is compute-bound and thus benefits the least from sharing. In terms of latency, all cases behave the same since the request execution flows are the same.

5.4 Communication-aware Placement

To show the effectiveness of communication-aware microservice placement, we evaluate HCM on E3 without data-plane orchestrator. In this case, all microservices are stationary after placement. We avoid host starvation and microservice overload by constraining problematic microservices to the host.

Using 3×Type1-SmartNIC and all placement constraints of Service Fabric [40] (described in §3.3), we compare HCM with both simulated annealing and an integer linear program (ILP). HCM places the highest importance on minimizing microservice communication latency. Simulated annealing and ILP use a cost function with the highest weight on minimizing co-execution interference. Hence, HCM tries to co-schedule communicating microservices on proximate resources, while the others will spread them out. ILP attempts to find the best configuration, while simulated annealing approximates. Figure 6 shows that compared to simulated annealing and ILP, HCM improves energy efficiency by up to 35.2% and 22.0%, and reduces latency by up to 24.0% and 18.6%, respectively. HCM's short communication latency benefits outweigh interference from co-execution.

5.5 Energy Efficiency = Cost Efficiency

While SmartNICs benefit energy efficiency and thus potentially bring cost savings, can they compete with other forms of heterogeneous clusters, especially when factoring in the capital expense to acquire the hardware? In this experiment, we evaluate the cost efficiency, in terms of request throughput over total cost and time of ownership, of using SmartNICs for microservices, compared with four other clusters (see Table 4). Assuming that clusters are usually at peak utilization, we use the cost efficiency metric $\frac{Throughput \times T}{CAPEX + (Power \times T \times Electricity)}$, where $Throughput$ is the measured average throughput at peak utilization for each application, as executed by E3 on each cluster, T is elapsed time, $CAPEX$ is the capital expense to purchase the cluster including all hardware components ($), $Power$ is the elapsed peak power draw of the cluster (Watts), and $Electricity$ is the price of electricity ($/Watts). The cluster cost and power data is shown in Table 4 and we use the average U.S. electricity price [31] of $0.0733/kWh. Figure 10 reports results for three applications of very different points in the workload space, extrapolated over time of ownership by our cost efficiency metric.

We make three observations. First, in the long term (>1 year of ownership), cost efficiency is increasingly dominated by energy efficiency. This highlights the importance of energy efficiency for data center design, where servers are typically replaced after several years to balance CAPEX [12]. Second, when all microservices are able to run on a low power platform (NFV-FIN), both 4×Wimpy and 2×Type2-SmartNIC clusters are the most cost efficient. After 5 years, 4×Wimpy is 14.1% more cost efficient than 2×Type2-SmartNIC because of the lower power draw. Third, when a microservice application contains both compute and IO-intensive microservices (RTA-SHM, IOT-TS), the 2×Type2-SmartNIC cluster is up to 1.9× more cost efficient after 5 years of ownership than the next best cluster configuration (4×Beefy in both cases).

Table 5 presents the measured energy-efficiency, which shows cost efficiency in the limit (over very long time of ownership). We can see that 4×Wimpy is only 3% more energy efficient (but has lower CAPEX) than 2×Type2-SmartNIC for NFV-FIN. 2×Type2-SmartNIC is on average 2.37× more energy-efficient (but has higher CAPEX) than 1×SuperBeefy, which is the second-best cluster in terms of energy-efficiency.

5.6 Performance at Scale

We evaluate and discuss the scalability of E3 along three axes: 1. Mechanism performance scalability; 2. Tail-latency; 3. Energy-efficiency.

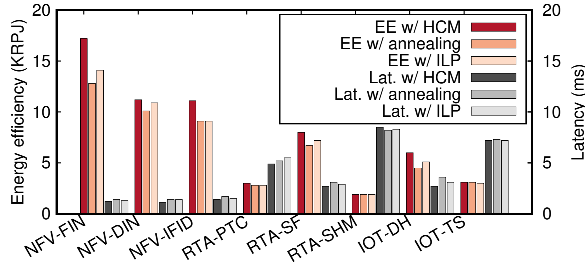


Figure 6: Communication-aware microservice placement.

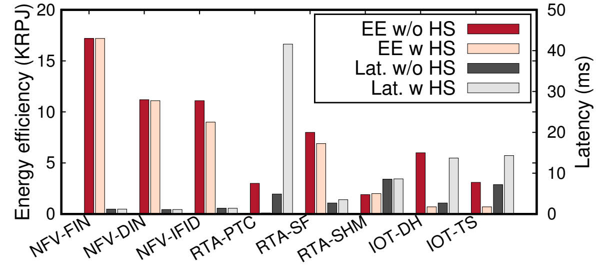


Figure 7: Avoiding host starvation (HS).

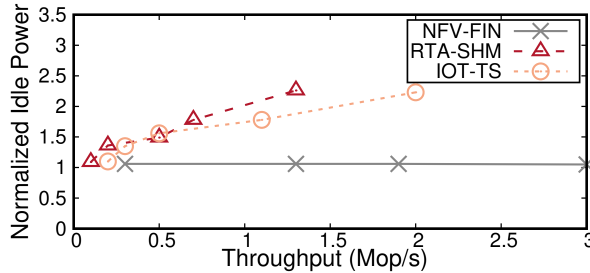


Figure 8: Power draw of 3 applications normalized to idle power of 3×Type1-SmartNIC, varying request load.

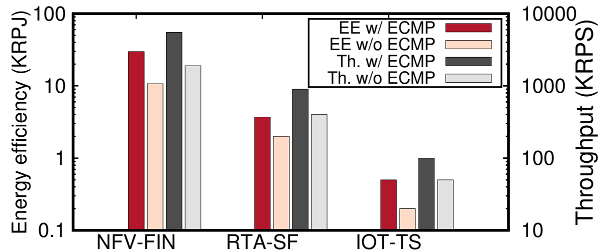
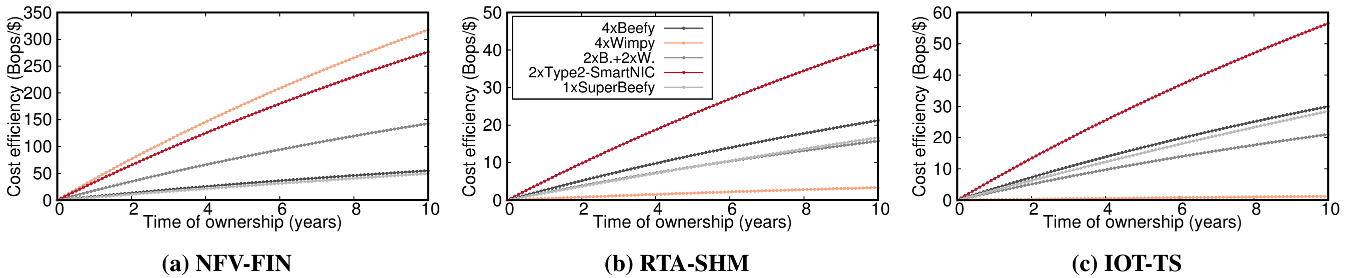


Figure 9: ECMP-based SmartNIC sharing (log y scale).



(a) NFV-FIN (b) RTA-SHM (c) IOT-TS

Figure 10: Cost efficiency of 3 applications across the cluster configurations from Table 4.

Servers →	100	200	400	600	800	1,000
HCM	4.85	8.31	19.83	34.32	74.39	263.46
Annealing	3.15	4.73	7.43	15.64	23.50	61.42
ILP	7.64	19.43	84.83	361.85	$\gg 1s$	$\gg 1s$

Table 6: Per-microservice deployment time (ms) scalability.

Mechanism scalability. At scale, pressure on the control-plane manager and data-plane orchestrator increases. We evaluate the performance scalability of both mechanisms with an increasing number of Type2-SmartNIC servers in a simulated FatTree [30] topology with 40 servers per rack. To avoid host starvation and microservice overload, E3's data-plane orchestrator receives one heartbeat message (16B) every 50ms from each SmartNIC that reports the queue length of the traffic manager and the SmartNIC's microservice execution times. The orchestrator parses the heartbeat message and makes a migration decision (§3.4). Figure 11 shows that the time taken to transmit the message and make a decision with a large number of servers stays well below the time taken to migrate the service (on the order of 10s-100s of ms) and is negligibly impacted by the number of deployed microservices. This is because the heartbeat message contributes only 1Kbps of traffic, even with 50K servers.

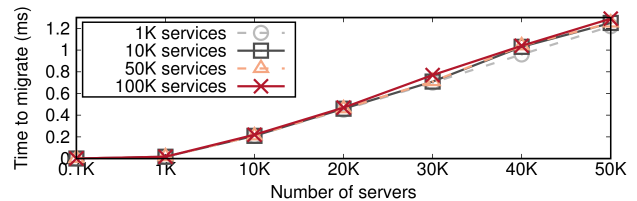


Figure 11: Orchestrator migration decision time scalability.

E3 uses HCM in the control-plane manager. We compare it to simulated annealing and ILP, deploying 10K microservices on an increasing number of servers. Table 6 shows that while HCM does not scale as well as simulated annealing, it can deploy new microservices in a reasonable time span (<<1s) at scale. ILP fails to deliver acceptable performance.

Tail latencies. At scale, tail latencies dominate [20]. While SmartNICs can introduce high tail latency for some microservices (§2.3), E3 places these microservices on the host to ensure that application-level tail-latency inflation is minimized

(§5.1). The tail-latency impact of SmartNIC offload is reduced at scale, as baseline communication latency increases with increasing inter-rack distance.

Energy-efficiency and power budgets. E3's energy efficiency benefits are constant regardless of deployment size and power budgets. At scale, there is additional energy cost for core and spine switches, but these are negligible compared to racks (ToRs and servers). Within a rack, ToR switch energy consumption stays constant, as all compared systems use the same number of switch ports. Our results show that E3 achieves up to 1.9x more throughput under the same power budget. Conversely, operators can save 48% of power, offering the same bandwidth.

6 Related Work

Architecture studies for microservices. Prior work has explored the architectural implications of running microservices [21, 27, 79]. It shows that wimpy servers hold potential for microservices under low loads, as they have less cache utilization and network virtualization overhead compared with traditional monolithic cloud workloads. FAWN [8] explored using only low-power processor architecture for data-intensive workloads as an energy and cost-effective alternative to server multiprocessors. However, FAWN assumed that I/O speeds are much lower than CPU speeds and so CPUs would be left idle for data-intensive applications. With the advent of fast network technologies, server CPUs are still required. Motivated by these studies, we focus on using SmartNIC-accelerated servers for energy efficiency.

Heterogeneous scheduling. A set of schedulers for performance-asymmetric architectures have been proposed. For example, Kumar et al. [44, 45] use instructions per cycle to determine relative speedup of each thread on different types of cores. HASS [73] introduces the architectural signature concept as a scheduling indicator, which contains information about memory-boundedness, available instruction-level parallelism, etc. CAMP [71] combines both efficiency and thread-level parallelism specialization and proposes a lightweight technique to discover which threads could use fast cores more efficiently. PTask [69] provides a data-flow programming model for programmers to manage computation for GPUs. It enables sharing GPUs among multiple processes, parallelizes multiple tasks, and eliminates unnecessary data movements. These approaches target long-running computations, mostly on cache coherent architectures, rather than microsecond-scale, request-based workloads over compute nodes that do not share memory and are hence not applicable.

Microservice scheduling. Wisp [76] enforces end-to-end performance objectives by globally adapting rate limiters and request schedulers based on operator policies under varying system conditions. This work is not concerned with SmartNIC heterogeneity. UNO [46] is an NFV framework that can systematically place NFs across SmartNIC and host with a resource-aware algorithm on the control plane. E3 is a microservice platform and thus goes several steps further: (1) E3 uses a data-plane orchestrator to detect node load and migrates microservices if necessary; (2) HCM considers communication distance during the placement. With the advent of SmartNICs and programmable switches, researchers have identified the potential performance benefits of applying request processing across the communication path [14]. E3 is such a system designed for the programmable cloud and explores the energy efficiency benefits of running microservices.

Power proportionality. Power proportional systems can vary energy use with the presented workload [50]. For example, Prekas, et al. propose an energy-proportional system management policy for memcached [65]. While E3 can provide energy proportionality, we are primarily interested in energy-efficiency. Geoffrey et al. [18] propose a heterogeneous power-proportional system. By carefully selecting component ensembles, it can provide an energy-efficient solution for a particular task. However, due to the high cost of ensemble transitions, we believe that this architecture is not fit for high bandwidth I/O systems. Rivoire, et al. propose a more balanced system design (for example, a low-power, mobile processor with numerous laptop disks connected via PCIe) and show that it achieves better energy efficiency for sorting large data volumes [67]. Pelican [11] presents a software storage stack on under-provisioned hardware targeted at cold storage workloads. Our proposal could be viewed as a balanced-energy approach for low-latency query-intensive server applications, rather than cold, throughput intensive ones.

7 Conclusion

We present E3, a microservice execution platform on SmartNIC-accelerated servers. E3 extends key system components (programming model, execution engine, communication subsystem, scheduling) of the Azure Service Fabric microservice platform to a SmartNIC. E3 demonstrates that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices.

Acknowledgments

This work is supported in part by NSF grants CNS-1616774, CNS-1714508, CNS-1751231 and the Texas Systems Research Consortium. We would like to thank the anonymous reviewers and our shepherd, Ada Gavrilovska, for their comments and feedback.

References

[1] DPDK. https://www.dpdk.org/.

[2] DynamIQ. https://developer.arm.com/technologies/dynamiq.

[3] Amazon Lambda Serverless Computing Platform. https://aws.amazon.com/lambda/, 2018.

[4] Google App Engine. https://cloud.google.com/appengine/, 2018.

[5] Nirmata Platform. https://www.nirmata.com/, 2018.

[6] Amazon. Amazon Data Pipeline. https://aws.amazon.com/datapipeline/, 2018.

[7] AMD. AMD HSA. https://www.amd.com/en-us/innovations/software-technologies/hsa, 2018.

[8] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[9] Microservice Architecture. Api gateway. http://microservices.io/patterns/apigateway.html, 2018.

[10] Microsoft Azure. Programming Model in the Azure Service Fabric. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework, 2018.

[11] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. Pelican: A Building Block for Exascale Cold Data Storage. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[12] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The Datacenter as a Computer: Designing Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2018.

[13] Broadcom. Broadcom Stingray SmartNICs. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2018.

[14] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. Beyond SmartNICs: Towards a fully programmable cloud. In *IEEE International Conference on High Performance Switching and Routing, ser. HPSR*, volume 18, 2018.

[15] Cavium. Cavium LiquidIO SmartNICs. https://cavium.com/pdfFiles/LiquidIO_II_CN78XX_Product_Brief-Rev1.pdf, 2018.

[16] Cavium. Cavium OCTEON Development Kits. https://cavium.com/octeon-software-develop-kit.html, 2018.

[17] Luis Ceze, Mark D Hill, and Thomas F Wenisch. Arch2030: A vision of computer architecture research over the next 15 years. *arXiv preprint arXiv:1612.03182*, 2016.

[18] Geoffrey Challen and Mark Hempstead. The Case for Power-agile Computing. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005.

[20] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, 2013.

[21] Christina Delimitrou. The Hardware-Software Implications of Microservices and How Big Data Can Help. 2018.

[22] A. Shehabi et al. United States Data Center Energy Usage Report. Technical report, Lawrence Berkeley National Laboratory, 2016.

[23] Dolphin Express. Remote Peer to Peer made easy. https://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf, 2018.

[24] 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*, 2013.

[25] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, 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*, 2018.

[26] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018.

[27] Y. Gan and C. Delimitrou. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[28] 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 Zaruvinsky, 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 & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[29] Google. Google Cloud Dataflow. https://cloud.google.com/dataflow/, 2018.

[30] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 54, pages 95–104, 2009.

[31] Site Selection Group. Power in the Data Center and its Cost Across the U.S. https://info.siteselectiongroup.com/blog/power-in-the-data-center-and-its-costs-across-the-united-states, 2017.

[32] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1998.

[33] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[34] HP. HPE ProLiant m800 Server Cartridge. https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04500667&sp4ts.oid=6532018, 2018.

[35] Thomas Hunter II. *Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Tooling*. 2017.

[36] Intel. Intel Xeon Phi Coprocessor 7120A. https://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1238-GHz-61-core, 2018.

[37] Rajkumar Jalan, Swaminathan Sankar, and Gurudeep Kamat. Distributed system to determine a server's health, 2018. US Patent 9,906,422.

[38] Brian Jeff. Ten Things to Know About big. LITTLE. *ARM Holdings*, 2013.

[39] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[40] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[41] 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*, 2015.

[42] 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 *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[43] 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*, 2016.

[44] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-ISA heterogeneous multi-core architectures: The

potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[45] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, 2004.

[46] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[47] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[48] Felix Xiaozhu Lin and Xu Liu. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[49] Jianxiao Liu, Zonglin Tian, Panbiao Liu, Jiawei Jiang, and Zhao Li. An approach of semantic web service classification based on Naive Bayes. In *Services Computing, 2016 IEEE International Conference on*, 2016.

[50] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Computer Architecture, 2014 ACM/IEEE 41st International Symposium on*, 2014.

[51] Mellanox. Mellanox BlueField Platforms. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_BlueField_Ref_Platform.pdf, 2018.

[52] MemSQL. In-memory SQL database. https://www.memsql.com, 2018.

[53] Microsoft. Azure IoT hub. https://azure.microsoft.com/en-us/services/iot-hub/, 2018.

[54] Microsoft. Honeywell Case Study. https://blogs.msdn.microsoft.com/azureservicefabric/2018/03/20/service-fabric-customer-profile-honeywell/, 2018.

[55] Microsoft. Microsoft Data Factory. https://azure.microsoft.com/en-us/services/data-factory/, 2018.

[56] Microsoft. NIC Teaming. https://docs.microsoft.com/en-us/windows-server/networking/technologies/nic-teaming/nic-teaming, 2018.

[57] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: A Data-centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[58] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[59] Netronome. Netronome Agilio SmartNIC. https://www.netronome.com/products/agilio-cx/, 2018.

[60] 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*, 2018.

[61] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[62] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3Sigma: Distribution-based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[63] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[64] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[65] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[66] 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*, 2018.

[67] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A Balanced Energy-efficiency Benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 2007.

[68] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[69] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[70] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[71] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, 2010.

[72] Felix Scholkmann, Jens Boss, and Martin Wolf. An efficient algorithm for automatic peak detection in noisy periodic and quasi-periodic signals. *Algorithms*, 5(4):588–603, 2012.

[73] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[74] SPEC. Trends in Server Efficiency and Power Usage in Data Centers. https://www.spec.org/events/beijing2016/slides/015-Trends_in_Server_Efficiency_and_Power_Usage_in_Data_Centers%20-%20Sanjay%20Sharma.pdf, 2016.

[75] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001.

[76] 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*, 2017.

[77] Mesh Systems. Mesh Systems. http://www.mesh-systems.com, 2018.

[78] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[79] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, 2016.

[80] Leendert van Doorn. Microsoft's datacenters. In *Proceedings of the 1st Workshop on Hot Topics in Data Centers*, 2016.

[81] Wikipedia. Aho-Corasick Algorithm. https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm, 2018.

[82] Wikipedia. Collaborative filtering. https://en.wikipedia.org/wiki/Collaborative_filtering, 2018.

[83] Wikipedia. Ema. https://en.wikipedia.org/wiki/Moving_average, 2018.

[84] Wikipedia. Equal-cost multi-path routing. https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing, 2018.

[85] Wikipedia. Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25, 2018.

[86] Wikipedia. Representational State Transfer Architecture. https://en.wikipedia.org/wiki/Representational_state_transfer, 2018.

[87] Wikipedia. k-nearest neighbors algorithm. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm, 2019.

[88] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU Sharing in NFV Systems. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive

Zhenyuan Ruan* Tong He Jason Cong
University of California, Los Angeles

Abstract

We present INSIDER, a full-stack redesigned storage system to help users fully utilize the performance of emerging storage drives with moderate programming efforts. On the hardware side, INSIDER introduces an FPGA-based reconfigurable drive controller as the in-storage computing (ISC) unit; it is able to saturate the high drive performance while retaining enough programmability. On the software side, INSIDER integrates with the existing system stack and provides effective abstractions. For the host programmer, we introduce virtual file abstraction to abstract ISC as file operations; this hides the existence of the drive processing unit and minimizes the host code modification to leverage the drive computing capability. By separating out the drive processing unit to the data plane, we expose a clear drive-side interface so that drive programmers can focus on describing the computation logic; the details of data movement between different system components are hidden. With the software/hardware co-design, INSIDER runtime provides crucial system support. It not only transparently enforces the isolation and scheduling among offloaded programs, but it also protects the drive data from being accessed by unwarranted programs.

We build an INSIDER drive prototype and implement its corresponding software stack. The evaluation shows that INSIDER achieves an average 12X performance improvement and 31X accelerator cost efficiency when compared to the existing ARM-based ISC system. Additionally, it requires much less effort when implementing applications. INSIDER is open-sourced [5], and we have adapted it to the AWS F1 instance for public access.

1 Introduction

In the era of big data, computer systems are experiencing an unprecedented scale of data volume. Large corporations like Facebook have stored over 300 PB of data at their warehouse, with an incoming daily data rate of 600 TB [62] in 2014. A recent warehouse-scale profiling [42] shows that data analytics has become a major workload in the datacenter. Operating on such a data scale is a huge challenge for system designers. Thus, designing an efficient system for massive data analytics has increasingly become a topic of major importance [23, 27].

The drive I/O speed plays an important role in the overall data processing efficiency—even for the in-memory computing framework [68]. Meanwhile, for decades the improvement of storage technology has been continuously pushing forward the drive speed. The two-level hierarchy (i.e., channel and bank) of the modern storage drive provides a scalable way to increase the drive bandwidth [41]. Recently, we witnessed great progress in emerging byte-addressable non-volatile memory technologies which have the potential to achieve near-memory performance. However, along with the advancements in storage technologies, the system bottleneck is shifting from the storage drive to the host/drive interconnection [34] and host I/O stacks [31, 32]. The advent of such a "data movement wall" prevents the high performance of the emerging storage from being delivered to end users—which puts forward a new challenge to system designers.

Rather than moving data from drive to host, one natural idea is to move computation from host to drive, thereby avoiding the aforementioned bottlenecks. Guided by this, existing work tries to leverage drive-embedded ARM cores [33, 57, 63] or ASIC [38, 40, 47] for task offloading. However, these approaches face several system challenges which make them less usable: 1) **Limited performance or flexibility**. Drive-embedded cores are originally designed to execute the drive firmware; they are generally too weak for *in-storage computing (ISC)*. ASIC, brings high performance due to hardware customization; however, it only targets the specific workload. Thus, it is not flexible enough for general ISC. 2) **High programming efforts.** First, on the host side, existing systems develop their own customized API for ISC, which is not compatible with an existing system interface like POSIX. This requires considerable host code modification to leverage the drive ISC capability. Second, on the drive side, in order to access the drive file data, the offloaded drive program has to understand the in-drive file system metadata. Even worse, the developer has to explicitly maintain the metadata consistency between host and drive. This approach requires a significant programming effort and is not portable across different file systems. 3) **Lack of crucial system support.** In practice, the drive is shared among multiple processes. Unfortunately, existing work assumes a monopolized scenario; the isolation and resource scheduling between different ISC tasks are not explored. Additionally, data protection is an important concern; without it, offloaded programs can issue arbitrary R/W requests to operate on unwarranted data.

To overcome these problems, we present INSIDER, a full-stack redesigned storage system which achieves the following design goals.

*Corresponding author.

Saturate high drive rate. INSIDER introduces the FPGA-based reconfigurable controller as the ISC unit which is able to process the drive data at the line speed while retaining programmability (§3.1). The data reduction or the amplification pattern from the legacy code are extracted into a drive program which could be dynamically loaded into the drive controller on demand (§3.2.2). To increase the end-to-end throughput, INSIDER transparently constructs a system-level pipeline which overlaps drive access time, drive computing time, bus data transferring time and host computing time (§3.5).

Provide effective abstractions. INSIDER aims to provide effective abstractions to lower the barrier for users to leverage the benefits of ISC. On the host side, we provide virtual file abstraction which abstracts ISC as file operations to hide the existence of the underlying ISC unit (§3.3). On the drive side, we provide a compute-only abstraction for the offloaded task so that drive programmers can focus on describing the computation logic; the details of underlying data movement between different system components are hidden (§3.4).

Provide necessary system support. INSIDER separates the control and data planes (§3.2.1). The control plane is trusted and not user-programmable. It takes the responsibilities of issuing drive access requests. By performing the safety check in the control plane, we *protect* the data from being accessed by unwarranted drive programs. The ISC unit, which sits on the data plane, only intercepts and processes the data between the drive DMA unit and storage chips. This compute-only interface provides an *isolated* environment for drive programs whose execution will not harm other system components in the control plane. The execution of different drive programs is hardware-isolated into different portions of FPGA resources. INSIDER provides an *adaptive drive bandwidth scheduler* which monitors the data processing rates of different programs and provides this feedback to the control plane to adjust the issuing rates of drive requests accordingly (§3.6).

High cost efficiency. We define cost efficiency as the effective data processing rate per dollar. INSIDER introduces a new hardware component into the drive. Thus, it is critical to validate the motivation by showing that INSIDER can achieve not only better performance, but also better cost efficiency when compared to the existing work.

We build an INSIDER drive prototype (§4.1), and implement its corresponding software stack, including compiler, host-side runtime library and Linux kernel drivers (§4.2). We could mount the PCIe-based INSIDER drive as a normal storage device in Linux and install any file system upon it. We use a set of widely used workloads in the end-to-end system evaluation. The experiment results can be highlighted as follows: 1) INSIDER greatly alleviates the system interconnection bottleneck. It achieves 7X∼11X performance compared with the host-only traditional system (§5.2.1). In most cases, it achieves the optimal performance (§5.2.2). 2) INSIDER achieves 1X∼58X (12X on average) performance and 2X∼150X (31X on average) cost efficiency compared to the ARM-based ISC system (§5.5). 3) INSIDER only requires moderate programming efforts to implement applications (§5.2.3). 4) INSIDER simultaneously supports multiple offloaded tasks, and it can enforce resource scheduling adaptively and transparently (§5.3).

2 Background and Related Work

2.1 Emerging Storage Devices: Opportunities and Challenges

Traditionally, drives are regarded as a slow device for the secondary persistent storage, which has the significantly higher access latency (in ms scale) and lower bandwidth (in hundreds of MB per second) compared to DRAM. Based on this, the classical architecture for storage data processing presented in Fig. 3a has met users' performance requirements for decades. The underlying assumptions of this architecture are: 1) The interconnection performance is higher than the drive performance. 2) The execution speeds of host-side I/O stacks, including the block device driver, I/O scheduler, generic block layer and file system, are much faster than the drive access. While these were true in the era of the hard-disk drive, the landscape has totally changed in recent years. The bandwidth and latency of storage drives have improved significantly within the past decade (see Fig. 1 and Fig. 2). However, meanwhile, the evolution of the interconnection bus remains stagnant: there have been only two updates between 2007 and 2017.[1]

For the state-of-the-art platform, PCIe Gen3 is adopted as the interconnection [66], which is at 1 GB/s bidirectional transmission speed per link. Due to the storage density[2] and due to cost constraints, the four-lane link is most commonly used (e.g., commercial drive products from Intel [7] and Samsung [14]), which implies the 4 GB/s duplex interconnection bandwidth. However, this could be easily transcended by the internal bandwidth of the modern drive [24, 33, 34]. Their internal storage units are composed of multiple channels, and each channel equips multiple banks. Different from the serial external interconnection, this two-level architecture is able to provide scalable internal drive bandwidth—a sixteen-channel, single-bank SSD (which is fairly common now) can easily reach 6.4 GB/s bandwidth [46]. The growing mismatch between the internal and external bandwidth prevents us from fully utilizing the drive performance. The mismatch gets worse with the advent of 3D-stacked NVM-based storage which can deliver comparable bandwidth with DRAM [35, 54]. On the other hand, the end of Dennard scaling slows down the performance improvement of CPU, making it unable to catch the ever-increasing drive speed. The long-established block layer is now reported to be a major

[1] Although the specification of PCIe Gen 4 was finalized at the end of 2017, there is usually a two-year waiting period for the corresponding motherboard to be available in the market. Currently there is no motherboard supporting PCIe 4.0, and we do not include it in the figure.

[2] CPU has limited PCIe slots (e.g., 40 lanes for an Xeon CPU) exposed due to the pin constraint. Using more lanes per drive leads to low storage density. In practice, a data center node equips 10 or even more storage drives.

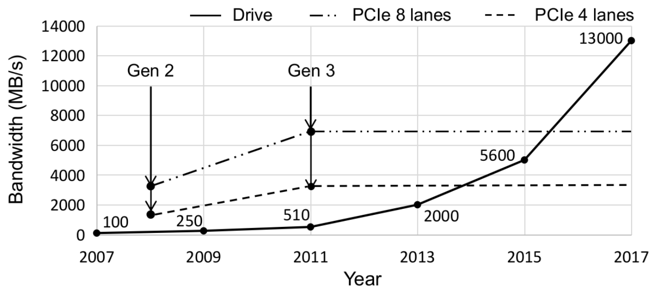


Figure 1: The bandwidth evolution of storage drives. Data are taken from [18] [4] [1] [8] [13] [16] in chronological order. This figure also presents the bandwidth evolution of PCIe (in 4 lanes and 8 lanes).

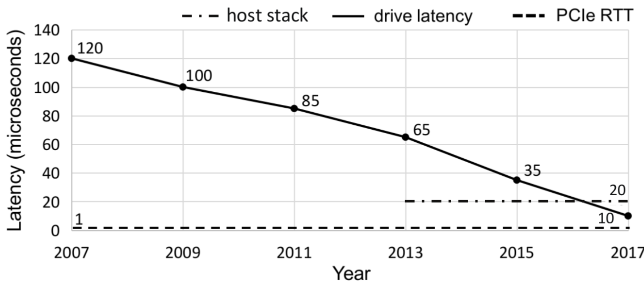


Figure 2: The latency evolution of storage drives. Data are taken from [15] [3] [9] [8] [10] [11] in chronological order. Meanwhile the latency of the host storage stack is about 20 μs [32], and the PCIe RTT (which includes the latency of bus and controller) is about 1 μs [50].

bottleneck of the storage system [28], and less than half raw drive speed is delivered to the end user [31, 56].

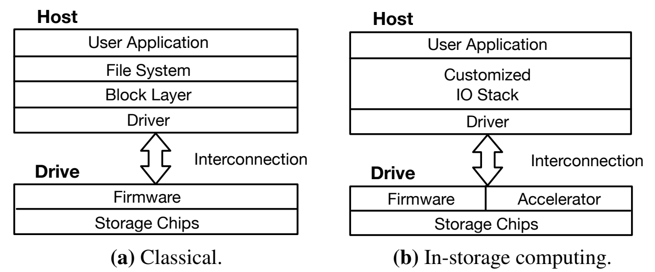
On the latency side, the state-of-the-art SSD delivers R/W latency below 10 μs [14], and the future NVM-based storage can potentially deliver sub-microsecond latency [30]. Meanwhile, the round-trip latency of PCIe still remains at about 1 μs [50], and the host-side I/O stack latency is even more than 20 μs [31, 32]. This implies that the latencies of host-side I/O stack are going to dominate the end-to-end latency.

In summary, the emerging storage devices bring hope—along with great challenges—to system designers. Unless the "data movement wall" is surpassed, high storage performance will not be delivered to end users.

2.2 Review of In-Storage Computing

In order to address the above system bottlenecks, the *in-storage computing (ISC)* architecture is proposed [48, 61], shown in Fig. 3b. In ISC, the host partially offloads tasks into the *in-storage accelerator* which can take advantage of the higher internal drive performance but is relatively less powerful compared to the full-fledged host CPU. For tasks that contain computation patterns like filtering or reduction, the output data volume of the accelerator, which will be transferred back to host via interconnection, is greatly reduced so that bottlenecks of interconnection and host I/O stacks are alleviated [33, 57, 63]. With customized IO stacks, the system bypasses the traditional OS storage stacks to achieve lower latency. With ISC, considerable performance and energy gains are achieved [25].

Historically, the idea of ISC was proposed two decades



(a) Classical. **(b)** In-storage computing.

Figure 3: Drive data processing architecture.

ago [43, 59], but did not become popular at that time. The reasons are twofold: 1) For the technology at that time, it was too expensive to integrate computing unit with storage drive; 2) More importantly, the drive performance was much lower than the performance of the host/drive bus, so in-storage computing could only bring limited performance benefits.

However, with the great improvement of VLSI technology in the past two decades, integration expense is greatly reduced. In fact, currently, every high-end SSD equips one or even multiple embedded CPUs. Meanwhile, the drive performance consistently increases, and goes beyond the performance of host/drive interconnection (see Fig. 1 and Fig. 2). This gap validates the motivation of ISC. Therefore, in recent years, we witness the revival of in-storage computing [49]. Most of the recent work focuses on offloading user-defined tasks to drive-embedded CPUs, which are originally designed to execute the firmware code, e.g., flash translation layer (FTL). However, this approach faces the following limitations.

Limited computing capability. Drive-embedded CPUs are usually fairly weak ARM cores which can be up to 100X slower compared to the host-side CPU (Table 3 in [63]). Based on this, offloading tasks to drive may lead to a decreased data processing speed by a factor of tens [33]. A recent work [48] proposes a dynamic workload scheduler to partition tasks between host and drive ARM processor. However, the optimal point they found is very close to the case in which all the tasks are executed at the host; this emphasizes that embedded cores are too feeble to provide a distinguishable speedup.

No effective programming abstractions. Existing work does not provide effective abstractions for programmers. On the host side, they develop their own customized API for ISC which is not compatible with an existing system interface like POSIX. This requires considerable host code modification to leverage the drive ISC capability. On the drive side, the drive program either manages the drive as a bare block device without a file system (FS), e.g., [48], or has to carefully cooperate with the host FS to access the correct file data, e.g., [32]. This distracts drive programmers from describing the computing logic and may not be portable across different FSes. It is important to provide effective abstractions to lower the barrier for users to leverage the benefits of ISC [26].

Lack of crucial system support. Naturally, the drive is shared among multiple processes, which implies the scenario of concurrently executing multiple ISC applications. This is

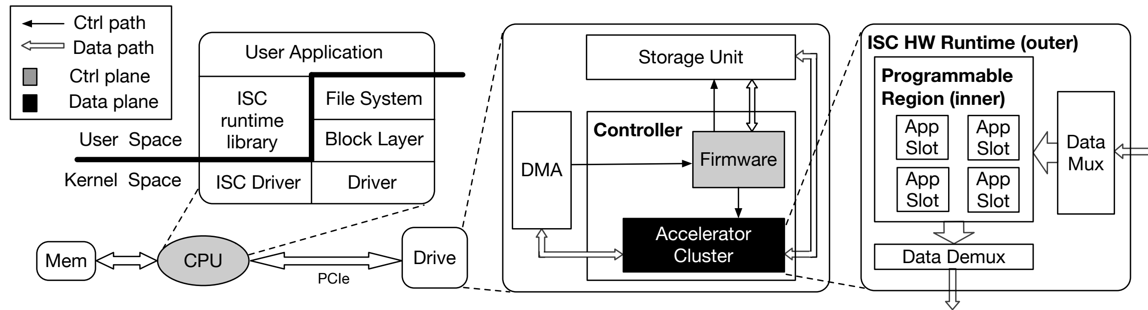


Figure 4: System architecture of INSIDER. INSIDER separates the control plane and the data plane; accelerator cluster sits on the data plane (black box) while the host-side library and drive-side firmware sit on the control plane (gray box).

especially important for the emerging storage drive since a single application may not fully saturate the high drive speed. It is crucial to provide support for protection, isolation and bandwidth scheduling. Without data protection, the malicious or erroneous ISC task may operate on unwarranted data; without isolation, the execution of one ISC task may harm the execution of other ISC tasks, or even the firmware execution; without bandwidth scheduling, some ISC tasks may forcibly occupy the drive, hampering fairness and liveness. However, existing work, e.g., [33, 34, 48], does not respond to these issues by assuming a monopolized execution environment.

Finally, another line of research equips the drive with an ASIC, which is the customized hardware chip designed for specific workloads. For instance, YourSQL [40] and Biscuit [38] equip a hardware IP with a key-based pattern matcher; work in [47] adopts a special hardware for database join and scan operations, etc. While ASIC-based solutions can achieve even much better performance compared to the high-end host CPU in their targeting applications, they are too specific to support other tasks. It requires the design of N chips to support N different applications; this introduces N times manufacturing, area size and energy cost. Thus, ASIC solutions are too inflexible to support general ISC.

3 INSIDER System Design

To overcome the problems above, we redesign the storage system across all layers, from the user layer down to the hardware layer. The design of INSIDER is introduced below.

3.1 FPGA-Based ISC Unit

The scenario of ISC puts forth several requirements to the in-drive processing unit.

High reconfigurability. As mentioned earlier, ASIC-based solutions can only target specific workloads. We wish the processing unit to be flexible enough to support general ISC.

Support massive parallelism. We analyze the computation patterns of data analytic workloads (§5.2) that are suitable for ISC. These applications expose abundant pipeline-level and data-level parallelism. The processing unit should have a proper architecture to capture those inherent parallelisms.

High energy efficiency. The storage drive is an energy-efficient device whose power consumption is just about 10 W [14]. The processing unit should not significantly compromise the energy efficiency of the drive.

	GPU	ARM	X86	ASIC	FPGA
Programmability	Good	Good	Good	No	Good
Data-level parallelism	Good	Poor	Fair	Best	Good
Pipeline-level parallelism	No	No	No	Best	Good
Energy efficiency	Fair	Fair	Poor	Best	Good

Table 1: Evaluating five candidates of ISC unit.

Given those requirements, we evaluate several candidates of ISC unit (see Table 1). FPGA comes out to be the best fit in our scenario. First, FPGA is generally reconfigurable and can form customized architectures for the targeted workloads. Second, through customization, FPGA can efficiently capture the inherent parallelism of applications. The data-level parallelism can be seized by replicating the processing elements to construct SIMD units [69]; the pipeline-level parallelism can be leveraged by constructing a deep hardware pipeline [60]. Finally, FPGA could achieve high energy efficiency between microprocessors and ASICs [58].

3.2 Drive Architecture

Fig. 4 presents the system architecture of INSIDER. We focus on introducing the drive-side design in this subsection.

3.2.1 Separating Control and Data Planes

The INSIDER drive controller consists of two decoupled components: the firmware logic and the accelerator cluster (i.e., the FPGA-based ISC unit). The firmware cooperates with the host-side ISC runtime and the ISC driver to enforce the *control plane* execution (marked in Fig. 4). It receives the incoming drive access requests from host, converts their logical block addresses into physical block addresses, and finally issues the requests to the storage unit. The accelerator cluster is separated out into the *data plane*. It does not worry about where to read (write) data from (to) the storage chip. Instead, it **intercepts and processes** the data between the DMA controller and the storage chip.

By separating control and data plane, we expose a *compute-only* abstraction for the in-drive accelerator. It does not proactively initiate the drive accessing request. Instead, it only passively processes the intercepted data from other components. The control plane takes the responsibilities of conducting file permission check at host and issuing drive accessing requests;

it prevents the drive data from being accessed by unwarranted drive programs. In addition, the compute-only abstraction brings an *isolated* environment for the accelerator cluster; its execution will not harm the execution of other system components in the control plane. The execution of different offloaded tasks in the accelerator cluster is further *hardware-isolated* into different portions of FPGA resources.

3.2.2 Accelerator Cluster

As shown in the rightmost portion of Fig. 4, the accelerator cluster is divided into two layers. The inner layer is a programmable region which consists of multiple application slots. Each slot can accommodate a user-defined application accelerator. Different than the multi-threading in CPU, which is time multiplexing, different slots occupy different portions of hardware resources simultaneously, thus sharing FPGA in spatial multiplexing. By leveraging partial reconfiguration [44], host users can dynamically load a new accelerator to the specified slot. The number of slots and slot sizes are chosen by the administrator to meet the application requirements, i.e., number of applications executing simultaneously and the resource consumption of applications. The outer layer is the hardware runtime which is responsible for performing flow control (§3.5) and dispatching data to the corresponding slots (§3.6). The outer layer is set to be user-unprogrammable to avoid safety issues.

3.3 The Host-Side Programming Model

In this section we introduce *virtual file abstraction* which is the host-side programming model of INSIDER. A virtual file is fictitious, but pretends to be a real file from the perspective of the host programmer—it can be accessed via a subset of the POSIX-like file I/O APIs shown in Table 2. The access to virtual file will transparently trigger the underlying system data movement and the corresponding ISC, creating an illusion that this file does really exist. By exposing the familiar file interface, the effort of rewriting the traditional code into the INSIDER host code is negligible.

We would like to point out that INSIDER neither implements the full set of POSIX IO operations nor provides the full POSIX semantics, e.g., crash consistency. The argument here is similar to the GFS [37] and Chubby [29] papers: files provide a familiar interface for host programmers, and exposing a file-based interface for ISC can greatly alleviate the programming overheads. Being fully POSIX-compliant is not only expensive but also unnecessary in most use cases.

3.3.1 Virtual File Read

Listing 1 shows a snippet of the host code that performs virtual file read. We will introduce the design of virtual file read based on the code order. Fig. 5 shows the corresponding diagram.

System startup During the system startup stage, INSIDER creates a hidden mapping file *.USERNAME.insider* in the host file system for every user. The file is used to store the virtual file mappings (which will be discussed soon). For security concerns, INSIDER sets the owner of the mapping file to the corresponding user and sets the file permission to 0640.

```
// register a virtual file
string virt = reg_virt_file(real_path,acc_id);
// open the virtual file
int fd = vopen(virt.c_str(),O_RDONLY);
if (fd != -1) {
  // send drive program parameters (if there are any)
  send_params(fd, param_buf, param_buf_len);
  int rd_bytes = 0;
  // read virtual file
  while (rd_bytes = vread(fd, buf, buf_size)) {
    // user processes the read data
    process(buf, rd_bytes);
  }
  // close virtual file, release resources
  vclose(fd);
}
```

Listing 1: Host-side code of performing virtual file read.

1). int vopen(const char *path, int flags)
2). ssize_t vread(int fd, void *buf, size_t count)
3). ssize_t vwrite(int fd, void *buf, size_t count)
4). int vsync(int fd)
5). int vclose(int fd)
6). int vclose(int fd, size_t *rfile_written_bytes)
7). string reg_virt_file(string file_path, string acc_id)
8). string reg_virt_file(tuple<string, uint, uint> file_sg_list, string acc_id)
9). bool send_params(int fd, void *buf, size_t count)

Table 2: INSIDER host-side APIs. *vwrite, vsync* will be discussed in §3.3.2 while others will be discussed in §3.3.1.

Registration. The host program determines the file data to be read by the in-drive accelerator by invoking *reg_virt_file* (method 7 in Table 2); it takes the path of a real file plus an application accelerator ID, and then maps them into a virtual file. Alternatively, *reg_virt_file* (method 8) accepts a vector of <file name, offset, length> tuples to support the gather-read pattern.[3] This allows us to create the virtual file based on discrete data from multiple real files. During the registration phase, the mapping information will be recorded into the corresponding mapping file, and the specified accelerator will be programmed into an available slot of the in-drive reconfigurable controller. INSIDER currently enforces a simple scheduling policy: it blocks when all current slots are busy.

File open. After registration, the virtual file can be opened via *vopen*. The INSIDER runtime will first read the mapping file to know the positions of the mapped real file(s). Next, the runtime issues the query to the host file system to retrieve the accessing permission(s) and the ownership(s) of the real file(s). Then, the runtime performs the file-level permission check to find out whether the *vopen* caller has the correct access permission(s); in INSIDER, we regard the host file system and INSIDER runtime as trusted components, while the user programs are treated as non-trusted components. If it is an unauthorized access, *vopen* will return an invalid file descriptor. Otherwise, the correct descriptor will be returned, and the corresponding accelerator slot index (used in §3.6) will be

[3] Currently INSIDER operates drive data at the granularity of 64 B, therefore the offset and length fields have to be multiples of 64 B. It is a limitation of our current implementation rather than the design.

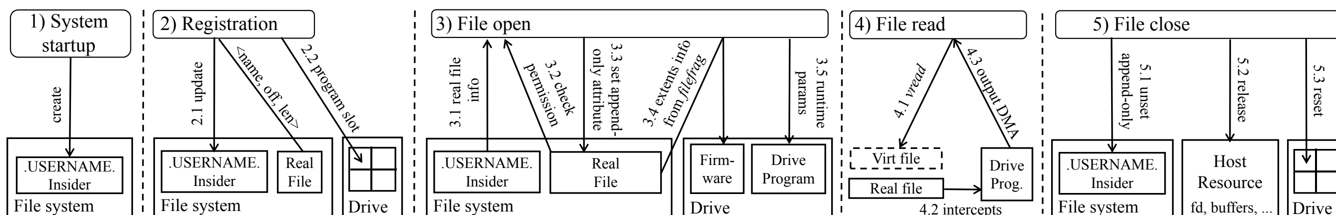


Figure 5: The system diagram of performing virtual file read. Only major steps are shown in this figure, see the text description in §3.3.1 for details.

sent to the INSIDER drive. After that, the INSIDER runtime asks the INSIDER kernel module to set the append-only attribute (if it is not already set by users before) on the mapped real file(s); this is used to guarantee that the current blocks of the real file(s) will not be released or replaced during the virtual file read.[4] Later on, INSIDER retrieves the locations of real file extents via the *filefrag* tool and transfers them to the drive. Finally, the host program sends runtime parameters of the accelerator program (discussed in §3.4), if there are any, via *send_params* to the drive.

File read. Now the host program can sequentially read the virtual file via *vread*. It first triggers the INSIDER drive to read the corresponding real file extents. The accelerator *intercepts* the results from the storage chips and invokes the corresponding data processing. Its output will be transferred back to the host via DMA, creating an illusion that the host is reading a normal file (which actually turns out to be a virtual file). The whole process is deeply pipelined without stalling. The detailed design of pipelining is discussed in §3.5. It seems to be meaningless to read the ISC results randomly, thus we do not implement a *vseek* interface.

File close. Finally, the virtual file is closed via *vclose*. In this step, the INSIDER runtime will contact the INSIDER kernel module to clear the append-only attribute if it was previously set in *vopen*. The host-side resource (e.g., file descriptor, the host-side buffer for DMA, etc.) will be released. Finally, the runtime sends the command to the INSIDER drive to reset the application accelerator to its initial state.

Virtual file read helps us to alleviate the bandwidth bottleneck in the drive → host direction. For example, for the feature selection application [64], the user registers a virtual file based on the preselected training data and the corresponding accelerator. The postselected result could be automatically read via *vread* without transferring the large preselected file from drive to host. Thus, the host program can simply use the virtual file as the input file to run the ML training algorithm.

3.3.2 Virtual File Write

Virtual file write works mostly in the same way but reverses the data path direction. We focus on describing the difference.

Registration. Virtual write requires users to preallocate enough space for the real file(s) to store the write output. If users leverage the *fallocate* system call to preallocate the file, they have to make sure to clear the *unwritten* flag on the file extents.[5] Otherwise, later updates on the real file may only be perceived via the INSIDER interface but not the OS interface.

File open. Besides the steps in §3.3.1, INSIDER runtime invokes $fsync$ to flush dirty pages of the real file(s) to drive if there are any. This guarantees the correct order between previous host-initiated write requests and the upcoming INSIDER drive-initiated write requests.

File write. In the file write stage, users invoke *vwrite* to write data to the virtual file. The written data is transferred to INSIDER drive through DMA, and then will be *intercepted and processed* by the accelerator. The output data will be written into the corresponding real file blocks. INSIDER also provides *vsync* (method 4 in Table 2), which can be used by users to flush in-core *vwrite* data to the INSIDER drive.

File close. Besides the steps in §3.3.1, INSIDER runtime will drop the read cache of the real file(s), if there are any, to guarantee that the newly drive-written data can be perceived by the host. This is conducted via calling *posix_fadvise* with *POSIX_FADV_DONTNEED*. Via invoking a variant of *vclose* (method 6 in Table 2), users can know the number of bytes written to the real file(s) by the underlying INSIDER drive. Based on the returned value, users may further invoke *ftruncate* to truncate the real file(s).

Virtual file write helps us alleviate the bandwidth bottleneck in the host → drive direction, since less data needs to be transferred through the bus (they then gets amplified in drive). For example, the user can register a virtual file based on a compressed real file and a decompression drive program. In this scenario, only compressed data needs to be transferred through the bus, and the drive performs in-storage decompression to materialize the decompressed file.

Since the virtual file write is mostly symmetric to the virtual file read, in the following we will introduce other system designs based on **the direction of read** to save space.

3.3.3 Concurrency Control

In INSIDER, a race condition might happen in the following cases: 1) Simultaneously a single real file is being *vwrite* and *vread*; 2) Simultaneously a real file is being *vwrite* by different processes; 3) A single real file is being *vread*, and meanwhile it is being written by a host program. In these cases, the users may encounter non-determinate results.

[4] With the append-only attribute, *ftruncate* will fail to release blocks, and the file defragmentation tool, e.g., *xfs_fsr* will ignore these blocks [21].

[5] In Linux, some file systems, e.g., ext4, will put the *unwritten* flag over the file extents preallocated by *fallocate*. Any following read over the extents will simply return zero(s) without actually querying the underlying drive; this is designed for security considerations since the preallocated blocks may contain the data from other users.

```
1  struct APP_Data {
2   char bytes[64]; // 64-byte width data interface.
3   unsigned short len; // Number of valid bytes.
4   bool eop; // Mark the end of the file.
5  };
6
7  // The ISC filter kernel.
8  void filter(Queue<APP_Data> &app_input_data,
9              Queue<APP_Data> &app_output_data,
10             Queue<unsigned int> &app_input_params) {
11  int lower_bound, upper_bound;
12  bool valid_lower_bound = false;
13  bool valid_upper_bound = false;
14  while (1) {
15   if (!valid_lower_bound) { // Read lower bound.
16    app_input_params.read(lower_bound);
17    valid_lower_bound = true;
18   } else if (!valid_upper_bound) { // Read upper bound.
19    app_input_params.read(upper_bound);
20    valid_upper_bound = true;
21   } else { // Do filtering
22    APP_Data record;
23    app_input_data.read(record);
24    // Call a macro (omitted in code) to transform the
25    // first 4B of input bytes into int.
26    int key = EXTRACT_KEY(record.bytes);
27    // Check the filtering condition.
28    if (lower_bound <= key && key <= upper_bound) {
29     // Write the matched record into the output queue.
30     APP_Data filtered_record;
31     // All 64 bytes data are valid.
32     filtered_record.len = 64;
33     // Set the EOP accordingly.
34     filtered_record.eop = record.eop;
35     // Copy the input record data.
36     for (int i = 0; i < 64; i ++) {
37      filtered_record.bytes[i] = record.bytes[i];
38     }
39     app_output_data.write(filtered_record);
40    } else if (record.eop) {
41     // Mark the EOP of the output when hitting input EOP.
42     filtered_record.len = 0;
43     filtered_record.eop = true;
44     app_output_data.write(filtered_record);
45    }
46   }
47  }
48 }
```

Figure 6: A simple example of the INSIDER drive accelerator code.

The problem also applies to Linux file systems: for example, different host processes may write to a same file. Linux file systems do not automatically enforce the user-level file concurrency control and leave the options to users. INSIDER makes the same decision here. When it is necessary, users can reuse the Linux file lock API to enforce the concurrency control by putting the R/W lock to the mapped real file.

3.4 The Drive-Side Programming Model

In this section we introduce the drive-side programming model. INSIDER defines a clear interface to hide all details of data movements between the accelerator program and other system components so that the device programmer *only* needs to focus on describing the computation logic. INSIDER provides a drive-side compiler which allows users to program in-drive accelerators with C++ (see Fig. 6 for a sample program). Additionally, the INSIDER compiler also supports the traditional RTL (e.g., Verilog) for experienced FPGA programmers. As we will see in §5.2, only C++ is used in the evaluation, and it can already achieve near-optimal performance in our scenario (§5.2.2).

Drive program interface consists of three FIFOs—data input FIFO, data output FIFO and parameter FIFO, as shown in the sample code. Input FIFO stores the intercepted data which is used for the accelerator processing. The output data of the accelerator, which will be sent back to host and acquired by *vread*, is stored into output FIFO. The host-sent runtime parameters are stored in parameter FIFO. The input and the output data are wrapped into a sequence of flits, i.e., *struct APP_Data* (see Fig. 6). The concept of flit is similar to the "word size" in host programs. Each flit contains a 64-byte payload, and the *eop* bit is used for marking the end of the input/output data. The length of data may not be multiples of 64 bytes, the *len* field is used to indicate the length of the last flit. For example, 130-byte data is composed by three flits; the last flit has *eop* = true and *len* = 2.

The sample program first reads two parameters, upper bound and lower bound, from the parameter FIFO. After that, in each iteration, the program reads the input record from the input FIFO. Then the program checks the filtering condition and writes the matched record into the output FIFO. Users can define stateful variables which are alive across iterations, e.g., line 11 - line 13 in Fig. 6, and stateless variables as well, e.g., line 22. These variables will be matched into FPGA registers or block RAMs (BRAMs) according to their sizes. The current implementation does not allow placing variables into FPGA DRAM, but it is trivial to extend.

INSIDER supports **modularity**. The user can define multiple sub-programs chained together with FIFOs to form a complete program, as long as it exposes the same drive accelerator interface shown above. Chained sub-programs will be compiled as separate hardware modules by the INSIDER compiler, and they will be executed in parallel. This is very similar to the dataflow architecture in the streaming system, and we can build a map-reduce pipeline in drive with chained sub-programs. In fact, most applications evaluated in §5.2 are implemented in this way. Stateful variables across sub-programs could also be passed through the FIFO interface.

3.5 System-Level Pipelining

Logically, in INSIDER, *vread* triggers the drive controller to fetch storage data, perform data processing, and transfer the output back to host. After that, the host program can finally start the host-side computation to consume the data. A naive design leads to the execution time $t = t_{drive_read} + t_{drive_comp.} + t_{output_trans.} + t_{host_comp}$. As we will see in §5.2, this leads to a limited performance.

INSIDER constructs a deep system-level pipeline which includes all system components involved in the end-to-end processing. It happens **transparently** for users; they simply use the programming interface introduced in §3.3 and §3.4. With pipelining, the execution time is decreased to $max(t_{drive_read}, t_{drive_comp.}, t_{output_trans.}, t_{host_comp})$.

Overlap $\mathbf{t_{drive_read}}$ with $\mathbf{t_{drive_comp.}}$ We carefully design the INSIDER hardware logic to ensure that it is fully pipelined, so that the storage read stage, computation stage and output DMA stage overlap one another.

Overlap drive, bus and host time We achieve this by ① Pre-issuing the file access requests during *vopen* which would trigger the drive to perform the precomputation; ② Allocating the host memory in the INSIDER runtime to buffer the drive precomputed results. With ①, the drive has all the position information of the mapped real file, and it can perform computation at its own pace. Thus, the host-side operation is decoupled from the drive-side computation. ② further decouples the bus data transferring from the drive-side computation. Now, each time that the host invokes *vread*, it simply pops the precomputed result from host buffers. To prevent the

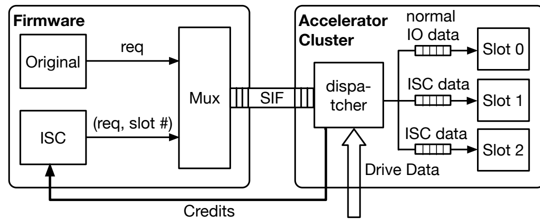


Figure 7: The drive architecture for supporting simultaneous multiple tasks.

drive from overflowing host buffers when host lags behind, INSIDER enforces credit-based flow control for each opened virtual file.

3.6 Adaptive Bandwidth Scheduler

Naturally, the drive is shared among multiple processes, which implies the scenario of parallel execution of multiple applications. For example, a single application may not fully saturate the high internal drive bandwidth so that the remaining bandwidth can be leveraged by others to improve the drive utilization. There are two concerns that should be addressed to support simultaneous multiple applications: 1) Given the fact that the drive is multiplexed among accelerators, we need a mechanism to dispatch drive data to the corresponding accelerator correctly. 2) Different accelerators have different data processing rates which can change dynamically. We need to implement a dynamic weighted fair queueing policy to schedule the drive bandwidth among accelerators adaptively.

Multiplexing. We develop a moderate extension to the original drive firmware (i.e., the one that does not support simultaneous multiple applications) to support multiplexing: we add an ISC unit and a multiplexer, see Fig. 7. The original firmware is used for handling the normal drive I/O requests as usual, while the ISC unit is used for handling the ISC-related drive requests. The ISC unit receives the file logical block addresses and the accelerator slot index from the INSIDER host runtime. The multiplexer will receive the request from both the ISC unit and the original firmware. The received request will be forwarded to the storage unit (not drawn in the figure), and its slot index will be pushed into the *Slots Index FIFO (SIF)*. The slot 0 is always locked for the pass-through logic, which is used for the normal drive read request since it does not need any in-storage processing. Thus, for the request issued by the original firmware, the MUX will push number 0 into SIF. After receiving the drive read data, the dispatcher of the accelerator cluster will pop a slot index from SIF and dispatch the data to the application FIFO connected to the corresponding application slot.

Adaptive Scheduling. The ISC unit maintains a set of *credit registers* (initialized to R) for all offloaded applications. The ISC unit will check registers of applications that have pending drive access requests, in a round-robin fashion. If the register of an application is greater than 0, the ISC unit will issue a drive access request in size C with its slot index, and then decrement its credit register. For the application with a higher data processing rate, its available FIFO size is going to be decreased more quickly, which brings us feedback information for performing the adaptive drive bandwidth scheduling. Each time data is dispatched to the application FIFO, the dispatcher will check the available space of that FIFO. If it is greater than C, the dispatcher will release one credit to the corresponding credit register in the ISC unit.

In practice, we choose the drive access request size C to be the minimal burst size that is able to saturate the drive bandwidth, and choose R to be large enough to hide the drive access latency. Ideally, we could solve an optimization problem to minimize the FPGA buffer size which equals to $R \cdot C$. We leave out the details here.

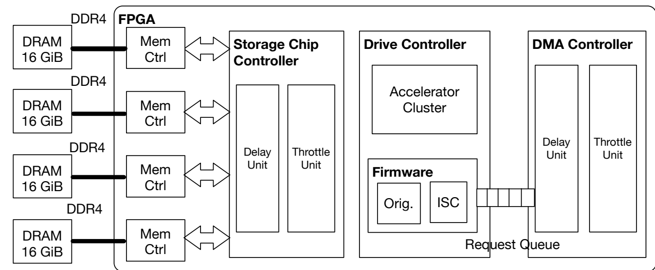


Figure 8: The diagram of the INSIDER drive prototype.

4 Implementation

The implementation of INSIDER contains 10K lines of C/C++ code (including the traditional host code and the FPGA HLS code), 2K lines of Verilog code and 2K lines of script code. The FPGA-side implementation is done based on the ST-Accel framework [60]. We have already adapted both the drive prototype and the software stack to the AWS F1 (FPGA) instance for public access, see [5].

4.1 The INSIDER Drive Prototype

So far there is no such a drive device on the market yet. We build an INSIDER drive prototype ourselves based on an FPGA board, see Fig. 8. The drive is PCIe-based and its implementation contains three parts: storage chip controller, drive controller and DMA controller. We implement a simple firmware logic in the drive controller; it is responsible for handling host-sent drive access commands, and the functionalities of performing wear-leveling and garbage collection have not been implemented yet. The remaining FPGA resource is used to accommodate the logics of drive programs. To emulate the performance of an emerging storage drive, our prototype should connect to multiple flash chips. However, there is no FPGA board in the market that has enough U.2 or M.2 flash connectors to meet our requirements. Therefore, in our prototype, we use four DRAM chips to emulate the storage chips. We add a set of delay and throttle units into the storage chip controller and DMA controller; they are configurable via our host-side API, therefore we could dynamically configure them to change the performance metrics (*i.e.,* bandwidth and latency) of our emulated storage chips and interconnection bus.

Application	Description	Comment	Data Size (GiB)	Parameter	Devel.Time (Person-Day)	LoC[6]	
						Host	Drive
Grep [38]	String matching.	Fully offloaded. Virtual file read.	60	983040 rows, 65536-byte row. 32-byte search string.	3	51	193
KNN [59]	K-nearest neighbors.	Offload the distance calculation into drive, and put K-partial sort in host. Virtual file read.	56.875	K = 32, 14680064 training cases, 4096 dims, 1-byte weight.	2	77	72
Bitmap file decompression	Decompress the bitmap file.	Offload run-length decoding into drive. Other preparation steps like header parsing and format checking are put in host. Virtual file write.	3.3	Compression ratio is about 7, width = 187577, height = 129000, planes = 1, depth = 8.	4	94	145
Statistics [55, 63]	Statistical calculation per input row.	Offload the row-level data reduction operations into drive, and put the computation over the reduced row data in host. Virtual file read.	48	65536 rows, 196608 numbers per row, 4-byte number.	3	65	170
SQL query [40, 65]	A query consists of select, sum, where, group by, and order by.	Offload data filtering into drive, and put sorting and grouping in host. Virtual file read.	60	2013265920 records 32-byte record.	5	97	256
Integration [48]	Combine data from different sources.	Fully offloaded. Virtual file read.	61	1006632960 records, 64-byte record, 32-byte query.	5	41	307
Feature Selection [64]	Relief algorithm to prune features.	Fully offloaded. Virtual file read.	61	15728640 hit records, 15728640 miss records, 256 dims.	9	50	632

Table 3: A descriptions of applications used in the evaluation. We present the experimental data sizes and application parameters. Additionally, we show the developing effort by listing the lines of code and the development time.

4.2 The INSIDER Software Stack

This section briefly introduces the software stack of INSIDER. We have omitted the details due to space constraints.

Compiler. INSIDER provides compilers for both host and drive. The host-side compiler is simply a standard *g++* which, by default, links to the INSIDER runtime. The front-end of the drive-side compiler is implemented on LLVM, while its back-end is implemented on Xilinx Vivado HLS [22] and a self-built RTL code transformer.

Software Simulator. FPGA program compilation takes hours, which greatly limits the iteration speed of the program development. INSIDER provides a system-level simulator which supports both the C++-level and RTL-level simulation. The simulator reduces the iteration time from hours into (tens of) minutes.

Host-side runtime library. The runtime library bypasses the OS storage stack and is at the user space. When necessary, it will use the POSIX I/O interface to interact with the host file system. Its implementation contains the drive parameter configuring API plus all methods in Table 2. Additionally, the runtime library cooperates with the drive hardware to support the system-level pipelining and the credit-based flow control.

Linux kernel drivers. INSIDER implements two kernel drivers. The first driver registers the INSIDER drive as a block device in Linux so that it could be accessed as a normal storage drive. The second driver is ISC related: it manages the DMA buffer for virtual file operations and is responsible for setting/unsetting the append-only attribute to the real file(s) in *vopen*/*vclose*.

5 Evaluation

5.1 Experiment Setup

We refer to the performance metrics of the current high-end SSDs to determine the drive performance used in our evaluation. On the latency side, the current 3D XPoint SSD already achieves latency less than 10 μs [6, 39]. On the throughput side, the high-end SSD announced in 2017 [17] could achieve

Host	Operating System	Linux LTS 4.4.169
	RAM	128 GB
	CPU	2*Intel Xeon E5-2686 v4
	File System	XFS
Drive	FPGA	Xilinx Virtex XCVU9P
	Capacity	64 GB
	Latency	5 μs
	Sequential R/W	16 GB/s (i.e., 14.9 GiB/s)
	Random 4K R/W	1200 KOPS
	Host/Drive Int. Speed	PCIe Gen3 x8 *or* x16
	Number of Slots	3

Table 4: Experiment setup.

13.0 / 9.0 GB/s sequential R/W performance. We project these numbers (according to the trend in Fig. 1, 2) to represent the performance of the next-generation high-performance drive. Table 4 provides details of our experiment setup. We use 32 CPU cores in the evaluation.

5.2 Applications

We choose applications used in the existing work to evaluate the INSIDER system (see Table 3). We implement them by ourselves. All drive programs are implemented in C++.

5.2.1 Speedup and Its Breakdown

See Fig. 9 for the performance comparison of seven applications. We choose the traditional host-only implementation which uses the POSIX interface as the baseline. It uses OpenMP to parallelize the computation to take advantage of 32 CPU cores. The first optimization is to replace the POSIX interface with the ISC interface to bypass overheads of the host I/O stack. This is conducted by registering the virtual file based on the real file and the pass-through (PT) program. The PT program simply returns all the inputs it receives as outputs. Thus, by invoking *vread* over the virtual file, we acquire the data of the real file. In Fig. 9, *Host-bypass* is the abbreviation for this version, while the suffix x8 and x16 stand for using PCIe Gen3 x8 and x16 as the interconnection, respectively. With the host-side code refactoring, we can conduct pipelining to overlap the computation time and the file access time;

[6]It does not include empty lines, comments, logging, timer, etc.

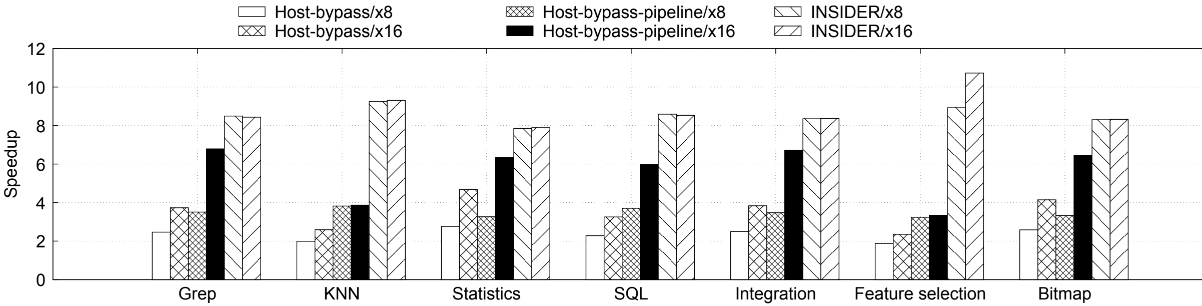
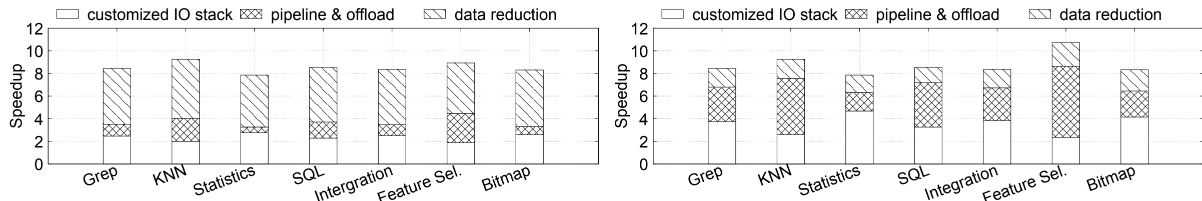


Figure 9: Speedup of optimized host-only versions and INSIDER version compared to the host-only baseline (§5.2.1).



(a) INSIDER/*x8* (*i.e,* the bus-limited case).

(b) INSIDER/*x16* (*i.e,* the bus-ample case).

Figure 10: The breakdown of the speedup achieved by INSIDER compared with the host-only baseline (§5.2.1).

this corresponds to *Host-bypass-pipeline* in Fig. 9. Finally, we leverage the ISC capability to offload computing tasks to the drive. For this version we largely reuse code from the baseline version since the virtual file abstraction allows us to stay at the traditional file accessing interface (§3.3) and INSIDER transparently constructs the system-level pipeline (§3.5). This corresponds to INSIDER in Fig. 9.

Note that the end-to-end execution time here includes the overheads of INSIDER APIs like *vopen, vclose*, but it does not include the overhead of reconfiguring FPGA, which is in the order of hundreds of milliseconds and is proportional to the region size [67]. We envision that in practice the application execution has time locality so that the overheads of reconfiguring will be amortized by multiple following calls.

The speedup of version INSIDER is derived from three aspects: 1) customized I/O stack (§4.2), 2) task offloading (§3.4) and system-level pipelining (§3.5), and 3) reduced data volume (which leads to lower bus time). See Fig. 10 for the speedup breakdown in these three parts. In the x8 setting, which has lower bus bandwidth, data reduction is the major source of the overall speedup. By switching from x8 to x16, the benefit of data reduction decreases, which makes sense since now we use a faster interconnection bus. Nevertheless, it still accounts for a considerable speedup. Meanwhile, pipelining and offloading contribute to a major part of the speedup.

As we discussed in §2.1, four-lane (the most common) and eight-lane links are used in real life because of storage density and cost constraints. INSIDER/*x16* does not represent a practical scenario at this point. The motivation for showing both the results of x8 and x16 is to compare the benefits of data reduction in both bus-limited and bus-ample cases.

5.2.2 Optimality and Bottleneck Analysis

Table 5 shows the performance bottleneck of different execution schemes for seven applications. For *Host-bypass*, limited PCIe bandwidth is the major bottleneck for the overall performance. In contrast, after enabling the in-storage processing, even in the PCIe x8 setting, there is only one case in which PCIe becomes the bottleneck (see INSIDER/*x8*). For most cases in INSIDER, the overall performance is bounded by the internal drive speed, which indicates that the **optimal performance** has been achieved. For some cases, like KNN and feature selection, host-side computation is the performance bottleneck for *Host-bypass*. This is alleviated in INSIDER since FPGA has better computing capabilities for the offloaded tasks. For INSIDER, *SQL query* is still bottlenecked by the host-side computation of the non-offloaded part.

	Host-bypass/x8	*Host-bypass/x16*	INSIDER/*x8*	INSIDER/*x16*
Grep	PCIe	PCIe	Drive	Drive
KNN	PCIe	Comp.	Drive	Drive
Statistics	PCIe	PCIe	Drive	Drive
SQL query	PCIe	Comp.	Comp.	Comp.
Integration	PCIe	PCIe	Drive	Drive
Feature selection	Comp.	Comp.	PCIe	Drive
Bitmap decompression	PCIe	PCIe	Drive	Drive

Table 5: The end-to-end performance bottleneck of different executing schemes over seven different applications. Here *PCIe, Drive and Comp.* indicate that the bottleneck is PCIe performance, drive chip performance and the host-side computation performance, respectively (§5.2.2).

5.2.3 Development Efforts

Table 3 also presents the developing efforts of implementing these applications in terms of lines of code (column *LoC*) and the developing time (column *Devel. Time*). With virtual file abstraction, all host programs here only require less than half an hour to be ported to the INSIDER; The main development time is spent on implementing the drive accelerator which requires drive programmers to tune the performance. This time is expected to be reduced in the future with continuous improvements on the FPGA programming toolchain. Addi-

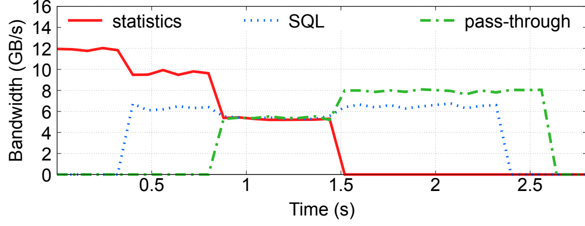


Figure 11: Data rates of accelerators that are executed simultaneously in drive. The drive bandwidth is 16 GB/s, and the bandwidth requested by *statistics*, *SQL* and *pass-through* are 12 GB/s, 6.4 GB/s and 8 GB/s, respectively. *statistics* starts before time 0 s and ends at about time 1.5 s. *SQL* starts at about time 0.4 s and ends at about time 2.4 s. *Pass-through* starts at about time 0.8 s and ends at about time 2.6 s.

	LUT	FF	BRAM	DSP
Grep	34416	24108	1	0
KNN	9534	11975	0.5	0
Statistics	14698	15966	0	0
SQL query	9684	14044	1	0
Integration	40112	6497	14	0
Feature selection	41322	44981	24	48
Bitmap decompression	60837	13676	0	0
INSIDER framework	68981	120451	309	0
DRAM and DMA IP cores	210819	245067	345.5	12
XCVU9P [19]	1181768	2363536	2160	6840
XC7A200T [2]	215360	269200	365	740

Table 6: The top half shows the FPGA resource consumption in our experiments. Generally, an FPGA chip contains four types of resources: look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs, which are SRAM-based), digital signal processors (DSPs). The bottom half shows the initial available resource in FPGA XCVU9P and XC7A200T.

tionally, since INSIDER provides a clear interface to separate the responsibilities between host and drive, drive programs could be implemented as a *library* by experienced FPGA developers. This can greatly lower the barrier for host users when it comes to realizing the benefits of the INSIDER drive.

Still, the end-to-end developing time is much less compared to an existing work. Table 1 in work [61] shows that WILLOW requires thousands of LoC and one-month development time to implement some basic drive applications like simple drive I/O (1500 LoC, 1 month) and file appending (1588 LoC, 1 month). WILLOW is definitely an excellent work, and here the main reason is that WILLOW was designed at a lower layer to extend the semantics of the storage drive, while INSIDER focuses on supporting ISC by exposing a compute-only interface at drive and file APIs at host.

5.3 Simultaneous Multiprocessing

In this section we focus on evaluating the effectiveness of the design in §3.6. We choose *statistics, SQL query, and pass-through* as our offloaded applications. On the drive accelerator side, we throttle their computing speeds below the drive internal bandwidth so that each of them cannot fully saturate the high drive rate: $BW_{drive} = 16$ GB/s, $BW_{stat} = 12$ GB/s, $BW_{SQL} = 6.4$ GB/s, $BW_{PT} = 8$ GB/s. The host-side task scheduling has already been enforced by the host OS, and our goal here is to evaluate the effectiveness of the drive-side bandwidth scheduling. Hence, we modify the host programs so that they only invoke INSIDER APIs without doing the host-side computation. In this case, the application execution time is a close approximation of the drive-side accelerator execution time. Therefore, the data processing rate for each accelerator can be calculated as $rate = \Delta size(data)/\Delta time$.

Fig. 11 presents the runtime data rate of three accelerators that execute simultaneously in drive. As we can see, INSIDER will try best to accommodate the bandwidth requests of offloaded applications. When it is not possible to do so, i.e., the sum of total requested bandwidth is higher than the drive bandwidth, INSIDER will schedule bandwidth for applications in a fair fashion.

5.4 Analysis of the Resource Utilization

Table 6 presents the FPGA resource consumption in our experiments. The end-to-end resource usage consists of three parts: ① User application logic. Row *Grep* to row *Bitmap decompression* correspond to this part. ② INSIDER framework. Row INSIDER *framework* corresponds to this part. ③ I/O IP cores. This part mainly comprises the resource for the DRAM controller and the DMA controller. Row *DRAM and DMA IP cores* correspond to this part.

We note that ③ takes the major part of the overall resource consumption. However, these components actually already exist (in the form of ASIC hard IP) in modern storage drives [33], which also have a built-in DRAM controller and need to interact with host via DMA. Thus, ③ only reflects the resource use that would only occur in our prototype due to our limited evaluation environment. The final resource consumption should be measured as ①+②. Row *XCVU9P* [19] and row *XC7A200T* show the available resource of a high-end FPGA (which is used in our evaluation) and a low-end FPGA [7], respectively. We notice that in the best case, the low-end FPGA is able to simultaneously accommodate five resource-light applications (*grep, KNN, statistics, SQL, integration*). The key insight here is that, for the ISC purpose, we *only* need to offload code snippet involving data reduction (related to the virtual file read) or data amplification (related to the virtual file write), therefore the drive programs are frugal in the resource usage.

5.5 Comparing with the ARM-Based System

Methodology. We assume that only the FPGA-based ISC unit is replaced by the ARM CPU, and all other designs remain unchanged. We extract the computing programs from the traditional host-only implementation used in §5.2. Since we assume the system-level pipelining (§3.5) is also deployed here, the final end-to-end time of the ARM-based platform could be calculated as $T_{e2e} = max(T_{host}, T_{trans}, T_{ARM})$, where T_{host} denotes the host-side processing time and T_{trans} denotes the host/drive data transferring time. Here, T_{host} and T_{trans} are taken from the measured data of INSIDER at §5.2. We target Cortex-A72 (using parameters in [12]), which is a high-end quad-core three-way superscalar ARM processor. We conduct runtime profilings over an ARM machine to extract

[7] We do not directly use XC7A200T in the evaluation since we cannot find a low-end FPGA board with large DRAM, which forces us to use XCVU9P.

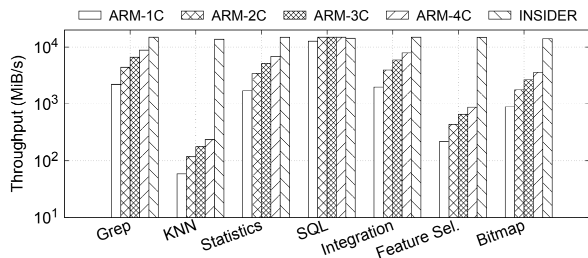


Figure 12: End-to-end data processing rates of INSIDER and the ARM-based platforms. ARM-*N*C means to use *N* core(s).

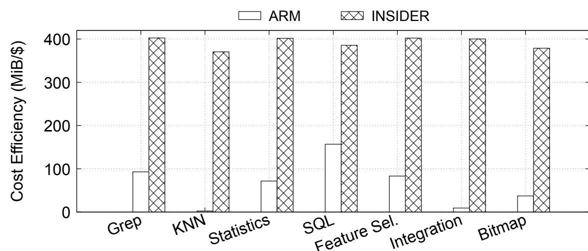


Figure 13: Cost efficiency (defined as data processing rates per dollar) of INSIDER and the ARM-based platforms. We do not include the cost of storage drive, whose price varies significantly across configurations.

the number of program instructions. The program execution time is then calculated *optimistically* by assuming that it has *perfect IPC* and *perfect parallelism* over multiple cores.

Fig. 12 (in log scale) shows the end-to-end data processing rates of INSIDER and the ARM-based platform. The speedup of INSIDER is highly related to the computation intensity of examined applications, but on average, INSIDER could achieve 12X speedup. For *KNN*, which is the most compute-intensive case, INSIDER could achieve 58X speedup; while for *SQL query*, which has the least computation intensity, the ARM-based platform could achieve the same performance.

We further present the cost efficiency of the ARM and INSIDER platforms, which is defined as the data processing rate per dollar. As discussed in §5.4, FPGA XC7A200T is already able to meet our resource demand; thus we use it in this evaluation. The wholesale price of FPGA is much less compared to its retail price according to the experience of Microsoft [36]. For a fair comparison, we use the wholesale prices of FPGA XC7A200T ($37 [20]) and ARM cortex-A72 ($95 [12]). We did not include the cost of storage drive in this comparison. Fig. 13 shows the cost efficiency results. Compared with the ARM-based platform, INSIDER achieves 31X cost efficiency on average. Specifically, it ranges from 2X (in *SQL query*) to 150X (in *KNN*).

6 Future Work

In-storage computing is still in its infancy. INSIDER is our initial effort to marry this architectural concept with a practical system design. There is a rich set of interesting future work, as we summarize in the following.

Extending INSIDER for a broader scenario. First, from the workload perspective, an extended programming model is desired to better support the data-dependent applications like key-value store. The current programming model forces host to initiate the drive access request, thus it cannot bypass the interconnection latency.

Second, from the system perspective, it would be useful to integrate INSIDER with other networked systems to reduce the data movement overheads. Compared to PCIe, performance of the network is further constrained, which creates yet another scenario for INSIDER [45]. The design of INSIDER is mostly agnostic to the underlying interconnection. By changing the DMA part into RDMA (or Ethernet), INSIDER can support the storage disaggregation case, helping cloud users to cross the "network wall" and take advantage of the fast remote drive. Other interesting use cases include offloading computation to HDFS servers and NFS servers.

Data-centric system architecture. Traditionally, the computer system is designed to be computing-centric, in which the data from IO devices are transferred and then processed by CPU. However, the traditional system is facing two main challenges. First, the data movement between IO devices and CPU has proved to be very expensive [53], which can no longer be ignored in the big data era. Second, due to the end of Dennard Scaling, general CPUs can no longer catch up with the ever-increasing speed of IO devices. Our long-term vision is to refactor the computer system into being data-centric. In the new architecture, CPU is only responsible for control plane processing, and it offloads data plane processing directly into the customized accelerator inside of IO devices, including storage drives, NICs [50, 52], memory [51], etc.

7 Conclusion

To unleash the performance of emerging storage drives, we present INSIDER, a full-stack redesigned storage system. On the performance side, INSIDER successfully crosses the "data movement wall" and fully utilizes the high drive performance. On the programming side, INSIDER provides simple but effective abstractions for programmers and offers necessary system support which enables a shared executing environment.

Acknowledgements

We would like to thank our shepherd, Keith Smith, and other anonymous reviewers for their insightful feedback and comments. We thank Wencong Xiao and Bojie Li for all technical discussions and valuable comments. We thank the Amazon F1 team for AWS credits donation. We thank Janice Wheeler for helping us edit the paper draft. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, the NSF NeuroNex award #DBI-1707408, and the funding from Huawei, Mentor Graphics, NEC and Samsung under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program. Zhenyuan Ruan is also supported by a UCLA Computer Science Departmental Fellowship.

References

[1] Anobit Announces Best-in-Class Flash Drives for Enterprise and Cloud Applications. https://www.businesswire.com/news/home/20110914005522/en/Anobit-Announces-Best-in-Class-Flash-Drives-Enterprise-Cloud.

[2] Artix-7 FPGA Product Table. https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html#productTable.

[3] CES 2009: pureSilicon 1TB Nitro SSD. https://www.slashgear.com/ces-2009-puresilicon-1tb-nitro-ssd-1230084/.

[4] DTS. http://www.storagesearch.com/dts.html.

[5] INSIDER Github Repository. https://github.com/zainryan/INSIDER-System.

[6] Intel Optane SDD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series/900p-280gb-aic-20nm.html.

[7] Intel Optane SSD DC P4800X. https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html.

[8] Intel Solid-State Drive 910 Series Product Specification. https://ark.intel.com/products/67009/Intel-SSD-910-Series-800GB-12-Height-PCIe-2_0-25nm-MLC.

[9] Intel X25-M 80GB SSD Drive Review. http://www.the-other-view.com/intel-x25.html.

[10] Lite-On SSD News. http://www.liteonssd.com/m/Company/news_content.php?id=LITE-ON-INTRODUCES-THE-NEXT-GENERATION-EP2-WITH-NVME-PROTOCOL-AT-DELL-WORLD-2015.html.

[11] Memory and Storage / Solid State Drives / Intel Enthusiast SSDs / Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series/900p-280gb-2-5-inch-20nm.html.

[12] Microprocessors - MPU QorIQ Layerscape. https://www.mouser.com/ProductDetail/NXP-Freescale/LS1046ASN8T1A?qs=sGAEpiMZZMup8ZLti7BNCxtNz7%252BF43hzZlkvLaqOJ8c%3D.

[13] Samsung Demos Crazy-Fast PCIe NVMe SSD At 5.6 GB Per Second At Dell World. https://hothardware.com/news/samsung-demos-crazy-fast-pcie-nvme-ssd-at-56-gb-per-second-showcases-16tb-ssd-at-dell-world.

[14] Samsung NVMe SSD 960 Pro. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-960-pro-m-2-512gb-mz-v6p512bw/.

[15] SanDisk:Solid State Disk Drive. https://www.anandtech.com/show/2151/4.

[16] Seagate announces 64TB NVMe SSD, Updated Nytro NVMe and SAS Lineup at FMS 2017. https://www.custompcreview.com/news/seagate-announces-64tb-nvme-ssd-updated-nytro-nvme-sas-lineup-fms-2017/.

[17] Seagate Nytro 5910 NVMe SSD. https://www.seagate.com/files/www-content/datasheets/pdfs/nytro-5910-nvme-ssdDS1953-4-1804US-en_US.pdf.

[18] Storage news - 2007, October week 3. http://www.storagesearch.com/news2007-oct3.html.

[19] UltraScale+ FPGAs Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[20] XC7A200T-1FFG1156C(IC Embedded FPGA Field Programmable Gate Array 500 I/O 1156FCBGA). https://www.alibaba.com/product-detail/XC7A200T-1FFG1156C-IC-Embedded-FPGA-Field_60730073325.html.

[21] XFS defragmentation tool will ignore the file which has append-only or immutable attribute set. https://kernel.googlesource.com/pub/scm/fs/xfs/xfsprogs-dev/+/v4.3.0/fsr/xfs_fsr.c#968.

[22] Xilinx Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[23] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 647–664, New York, NY, USA, 2017. ACM.

[24] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent SSD: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Conference on information and knowledge management*, CIKM '13, pages 1573–1576, New York, NY, USA, 2013. ACM.

[25] R. Balasubramonian and B. Grot. Near-Data Processing [Guest editors' introduction]. *IEEE Micro*, 36(1):4–5, Jan 2016.

[26] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 56–61, New York, NY, USA, 2017. ACM.

[27] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth European Conference on Computer Systems*, EuroSys '18, 2018.

[28] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[29] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[30] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[33] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[34] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[35] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. Phoenix: Memory speed hpc i/o with nvm. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 121–131, Dec 2016.

[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, 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, 2018. USENIX Association.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[38] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In

Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[39] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[40] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-performance Database System Leveraging In-storage Computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[41] 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*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.

[42] 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. ACM.

[43] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[44] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, 2018. USENIX Association.

[45] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[46] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory SSDs. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[47] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage Processing of Database Scans and Joins. *Inf. Sci.*, 327(C):183–200, January 2016.

[48] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[49] Philip Kufeldt, Carlos Maltzahn, Tim Feldman, Christine Green, Grant Mackey, and Shingo Tanaka. Eusocial Storage Devices: Offloading Data Management to Storage Devices that Can Act Collectively. *;login:*, 43(2), 2018.

[50] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[51] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 288–301, New York, NY, USA, 2017. ACM.

[52] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, pages 22–28, 2017.

[53] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 2019.

[54] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[55] Jian Ouyang, Shiding Lin, Zhenyu Hou, Peng Wang, Yong Wang, and Guangyu Sun. Active SSD Design for Energy-efficiency Improvement of Web-scale Data Analysis. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 286–291, Piscataway, NJ, USA, 2013. IEEE Press.

[56] 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*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[57] D. Park, J. Wang, and Y. S. Kee. In-Storage Computing for Hadoop MapReduce Framework: Challenges and Possibilities. *IEEE Transactions on Computers*, PP(99):1–1, 2016.

[58] A. Putnam. (Keynote) The Configurable Cloud - Accelerating Hyperscale Datacenter Services with FPGA. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1587–1587, April 2017.

[59] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[60] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. St-accel: A high-level programming platform for streaming applications on fpga. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, April 2018.

[61] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, 2014. USENIX Association.

[62] Cassidy R. Sugimoto, Hamid R. Ekbia, and Michael Mattioli. *Big Data Is Not a Monolith*. The MIT Press, 2016.

[63] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, San Jose, CA, 2013. USENIX.

[64] Ryan J Urbanowicz, Melissa Meeker, William LaCava, Randal S Olson, and Jason H Moore. Relief-based feature selection: introduction and review. *arXiv preprint arXiv:1711.08421*, 2017.

[65] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *PVLDB*, 7(11):963–974, 2014.

[66] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 6:1–6:11, New York, NY, USA, 2015. ACM.

[67] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 22:1–22:7, New York, NY, USA, 2017. ACM.

[68] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '18, 2018.

[69] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, Nov 2016.

Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval

Shengwen Liang[†,*], Ying Wang[†,*,1], Youyou Lu[‡], Zhe Yang[‡], Huawei Li[†,*,1], Xiaowei Li[†,*]
State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing[†],
University of Chinese Academy of Sciences[*], Tsinghua University[‡]

Abstract

Data analysis and retrieval is a widely-used component in existing artificial intelligence systems. However, each request has to go through each layer across the I/O stack, which moves tremendous irrelevant data between secondary storage, DRAM, and the on-chip cache. This leads to high response latency and rising energy consumption. To address this issue, we propose Cognitive SSD, an energy-efficient engine for deep learning based unstructured data retrieval. In Cognitive SSD, a flash-accessing accelerator named DLG-x is placed by the side of flash memory to achieve near-data deep learning and graph search. Such functions of in-SSD deep learning and graph search are exposed to the users as library APIs via NVMe command extension. Experimental results on the FPGA-based prototype reveal that the proposed Cognitive SSD reduces latency by 69.9% on average in comparison with CPU based solutions on conventional SSDs, and it reduces the overall system power consumption by up to 34.4% and 63.0% respectively when compared to CPU and GPU based solutions that deliver comparable performance.

1 Introduction

Unstructured data, especially unlabeled videos and images, etc., have grown explosively in recent years. It is reported that the unstructured data occupies up to 80% of storage capacity in commercial datacenters [10]. Once being stored and managed in the cloud machines, the massive amount of unstructured data leads to intensive retrieval requests issued by users, which pose significant challenge to the processing throughput and power consumption of a datacenter [19]. Consequently, it is critical to support fast and energy-efficient data retrieval in the cloud service infrastructure to reduce the total cost of ownership (TCO) of datacenters.

Unfortunately, conventional content-based multimedia data retrieval systems suffer from the issues of inaccuracy, power inefficiency, and high cost especially for large-scale unstructured data. Fig. 1(a) briefly depicts a typical content-based

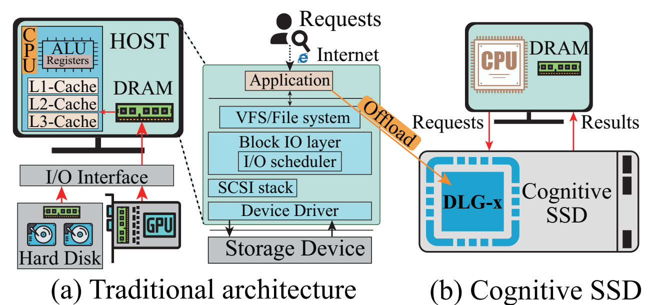


Fig. 1. Traditional architecture (a) vs. Cognitive SSD (b).

data retrieval system composed of CPU/GPU and conventional storage devices based on a compute-centric architecture [14]. When a data retrieval request arrives from the internet or the central server, the CPU has to reload massive potential data from disk into the temporary DRAM [14] and match the features of the query with those of the loaded unstructured data to find the relevant targets. This compute-centric architecture is confronted with several critical sources of overhead and inefficiency. (1) The current I/O software stack significantly burdens the data retrieval system when it simply fetches data from the storage devices on retrieval requests [60], as shown in Fig. 1(a). The situation is even worse since it is reported the performance bottleneck has migrated from hardware (75∼50us [11]) to software (60.8us [48]) as traditional HDDs are replaced by non-volatile memory [41,48]. (2) Massive data movement incurs energy and latency overhead in the conventional memory hierarchy. This issue becomes severe as the scale of data under query increases because the relevant data at the low-level storage must travel across a slow I/O interface (e.g., SATA), main memory and multi-level caches before reaching the compute units of CPU or GPUs [24], which is depicted in Fig. 1(a).

To address these issues, as shown in Fig. 1(b), this work aims to tailor a unified data storing and retrieval system within the compact storage device, and eliminate the major IO and data moving bottleneck. In this system, retrieval requests are directly sent to the storage devices, and the target data analysis and indexing are completely performed where the unstruc-

[1]Corresponding authors are Ying Wang and Huawei Li.

tured data resides. Building such a data retrieval system based on the proposed Cognitive SSD bears the following design goals: (1) providing a high accuracy, low latency, and energy efficient query mechanism affordable in a compact SSD, (2) exploiting the internal bandwidth of flash devices in an SSD for energy-efficient deep learning based data processing, and (3) enabling developers to customize the data retrieval system for different dataset. These points are stated in detail below.

First, instead of relying on the general-purpose CPU or GPU devices in Fig. 1(a), we must have a highly computation-efficient yet accurate data retrieval architecture in consideration of the SSD form factor and cost. A conventional data retrieval framework is inaccurate or too computationally expensive to be implemented within a resource-constrained SSD. In this work, we are the first to propose a holistic data retrieval mechanism by combining the deep learning and graph search algorithm (DLG), where the former could extract the semantic features of unstructured data and the latter could improve database search efficiency. The DLG solution achieves much higher data retrieval accuracy and enables user-definable computation complexity through deep learning model customization, making it possible to implement a flexible and efficient end-to-end unstructured data retrieval system in the SSD.

Second, although DLG is a simple and flexible end-to-end data retrieval solution, embedding it into SSDs still takes considerable effort. We designed a specific hardware accelerator that supports deep hashing and graph search simultaneously, DLG-x, to construct the target Cognitive SSD without using power-unsustainable CPU or GPU solutions. However, the limited DRAM inside an SSD is mostly used to cache the metadata for flash management, leaving no free space for the deep learning applications. Fortunately, we have proved that the bandwidth of internal flash interface surpasses that of external IO interface in a typical SSD, which matches the bandwidth demand of the DLG-x with proper data layout mapping. By rebuilding the data path in the SSD and deliberately optimizing the data-layout related to deep learning models and graphs on NAND flash, the DLG-x could fully exploit internal parallelism and directly access data from NAND flash bypassing the on-board DRAM.

Finally, as we introduce deep learning technology into the SSD, we must expose the software abstraction of Cognitive SSD to users and developers to process different data structures with different deep learning models. Thus, we abstract the underlying deep learning mechanism, feature analysis, and data structure indexing mechanism as user-visible calls by utilizing the NVMe protocol [6] for command extension. Not only can users' requests trigger the DLG-x accelerator to search the target dataset for query-relevant structures, but also system developers can freely configure the deep hashing architecture with different representation power and overhead for different dataset and performance requirement. In contrast to conventional ad-hoc solutions, Cognitive SSD allows system developers to adjust the retrieval accuracy as well as the real-time performance of the data retrieval service through provided APIs. Meanwhile, Cognitive SSD also supports the flexible combination of special commands to achieve different data retrieval related tasks, like in-storage data categorization and hashing-only functions. In summary, we make the following novel contributions:

1. We propose Cognitive SSD, to enable within-SSD deep learning and graph search by integrating a specialized deep learning and graph search accelerator (DLG-x). The DLG-x directly accesses data from NAND flash without crossing multiple memory hierarchies to decrease data movement path and power consumption. To the best of our knowledge, this work is the first to combine the deep learning and graph search methods for fast and accurate data retrieval in SSD.
2. We employ Cognitive SSD to build a serverless data retrieval system, which completely abandons the conventional data query mechanism in orthodox compute-centric systems. It can independently respond to data retrieval requests at real-time speed and low energy cost. It can also scale to a multi-SSD system and significantly reduces the hardware and power overhead of large-scale storage nodes in data centers.
3. We build a prototype of Cognitive SSD on the Cosmos plus OpenSSD platform [7] and use it to implement a data retrieval system. Our evaluation results demonstrate that Cognitive SSD is more energy-efficient than a multimedia retrieval system implemented on CPU and GPU, and reduces latency by 69.9% on average compared to the implementation with CPU. We also show that it outperforms conventional computing and storage node used in the data center when Cognitive SSD scales out to form smart lightweight storage nodes that include connected Cognitive SSD array.

2 Background and Preliminaries

2.1 Unstructured Data Retrieval System

Content-based unstructured data retrieval systems aim to search for certain data entries from the large-scale dataset by analyzing their visual or audio content. Fig. 2 depicts a typical content-based retrieval procedure consists of two main stages: feature extraction, and database indexing. Feature extraction generates the feature vector for the query data, and database indexing searches for similar data structures in storage with that feature vector encoded in a semantic space.

Feature Extraction and Deep Learning. The rise of deep learning transfers the focus of researches to deep convolution neural network (DCNN) [38] based features [61], as it provides better mid-level representations [37,40]. Fig. 2 depicts a typical DCNN that contains four key types of network layers: (1) convolution layer, which extracts visual feature from input by moving and convolving multidimensional filters across the input data organized into 3D tensors, (2) activation, the

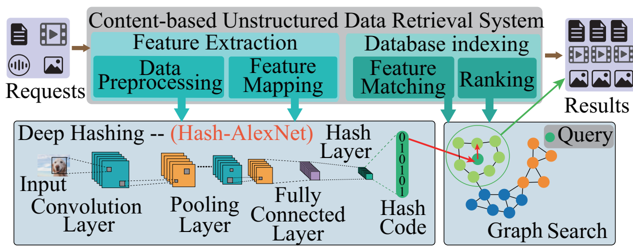


Fig. 2. Content-based multimedia data retrieval system.

nonlinear transformation that we do over the input signal, (3) pooling layers, which down-sample the input channels for scale and other types of invariance, and (4) fully connected (FC) layer, which performs linear operations between the features and the learned weights to predict the categorization or other high level characteristics of input data. Such a neural network is flexible and can be designed to have different hyper-parameters, like the number of the convolution and pooling layers stacked together and the dimension and number of convolution filters. Changing these parameters will impact the generalization ability and also computational overhead of neural networks, which are usually customizable for different dataset or application scenarios [47]. Some prior work directly employs the high-dimension output vector of the FC layer for data retrieval and is thought too expensive in terms of memory footprint and computation complexity [39]. Thus, we adopt deep hashing [38] to achieve effective yet condensed data representation. Fig. 2 exemplifies a deep hashing architecture, Hash-AlexNet, where a hash layer follows the last layer of AlexNet [35] to project the data feature learned from AlexNet into the hash space, and the generated hash code can be directly used to index the relevant data structures and get rid of the complex data preprocessing stage.

Database indexing: Graph-based approximate nearest neighbor search(ANNS) methods named NSG [27], a complement to deep hashing, achieves both accurate and fast data retrieval results, as was proved in previous work [17, 26]. The main idea of NSG is mapping the query hash code into a graph. The vertex of the graph represents an instance, and the edge stands for the similarities between entities, where the value of the edge represents the strength of similarity. On top of that, NSG can iteratively check neighbors' neighbors in the graph to find the true neighbors of the query based on the neighbor of a neighbor is also likely to be a neighbor concept. In this manner, the NSG could avoid unnecessary data checking to reduce retrieval latency.

In summary, deep hashing followed by graph search can perform low-latency and high-precision retrieval performance compared to traditional solutions using brute-force search or hash algorithms. Meanwhile, it also makes the retrieval framework more compact and efficient because of the similar compute patterns and data stream, so that they can fit into compact and power-limited SSDs.

2.2 Near data processing & deep learning accelerator

For hardware-software co-design, there are two directions in SSD research: open-channel SSD and near-data processing (NDP). While open-channel SSD enables direct flash memory access via system software [15, 42, 44, 59], near-data processing (NDP) moves computation from the system's main processors into memory or storage devices [16, 18, 25, 46, 50, 51, 56].

In NDP, Morpheus [52] provides a framework for moving computation to the general-purpose embedded processors on NVMe SSD. FAWN [13] uses low-power processors and flash to handle data processing and focuses on a key-value storage system. SmartSSD [34] introduces the Smart SSD model, which pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying the operating system. [53] supports fundamental database operations including sort, scan, and list intersection by utilizing Samsung SmartSSD. [23] investigates by simulation the possibility of employing the embedded ARM processor in SSDs to run SGDs, which is a key components of neural network training. However, none of them can handle deep learning processing due to the performance limit of the embedded processor. Thereby, [22] presents intelligent solid-state drives (iSSDs) that embed stream processors into the flash memory controllers to handle linear regression and k-means workloads. [43] integrates programmable logics into SSDs to achieve energy-efficiency computation for web-scale data analysis. Meanwhile, [32] also uses FPGAs to construct BlueDBM that uses flash storage and in-store processing for cost-effective analytics of large datasets, such as graph traversal and string search. GraFBoost [33] focuses on the acceleration of graph algorithms on an in-flash computing platform instead of deep learning algorithms as this work.

Cognitive SSD Prior active disks are integrating either general purpose processors incapable of handling high-throughput data or specialized accelerators with only the support of simple functions like scanning and sorting. These in-disk computation engines are unable to fulfill the requirement of high-throughput deep neural network (DNN) inference because computation-intensive DNNs generally rely on power-consuming CPU or GPUs in the case of data analysis and query tasks. To enable energy-efficient DNN, prior work proposes a variety of energy-efficient deep learning accelerators. For example, Diannao and C-Brain map large DNNs onto a vectorized processing array and employ a data tiling policy to exploit locality in neural parameters [20, 49]. Eyeriss applies the classic systolic array architecture to the inference of CNN, and outperforms CPU and GPU in energy efficiency dramatically [21]. However, these researches focus on optimizing the internal structure of accelerator and relied on large-capacity SRAM or DRAM instead of external non-volatile memory. In contrast to these works and prior active SSD designs, we propose **Cognitive SSD**, the first work that enables the storage device to employ deep learning to conduct

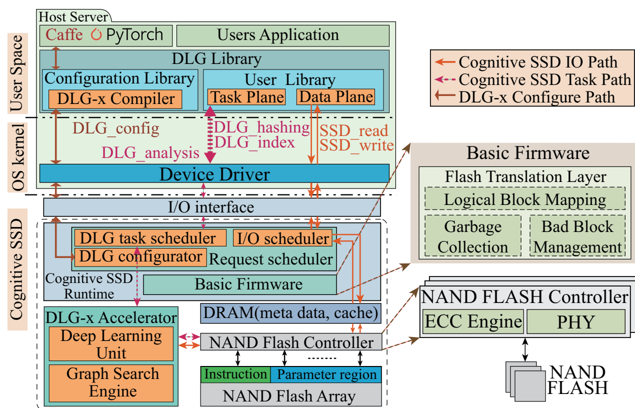


Fig. 3. Overview of the Cognitive SSD system.

in-storage data query and analysis. It is designed to replace the conventional data retrieval system and contains a flash-accessing accelerator (DLG-x) for deep learning and graph search. The DLG-x is deliberately reshaped to take advantage of the large flash capacity and high internal bandwidth, and it is also re-architected to enable graph search to target indexing.

3 Cognitive SSD System

Target Workload. As shown in Fig. 2, this work combines the strengths of deep hashing and graph search technique (DLG) to reduce the complexity of retrieval systems on the premise of high accuracy, which makes it possible to offload retrieval task from CPU/GPU into the resource-constraint Cognitive SSD. Based on that, we build an end-to-end data retrieval system that supports multimedia data retrieval such as audio, video and text. For example, audio can be processed by recurrent or convolutional neural network models on Cognitive SSD to generate hash codes, which act as an index for retrieving relevant audio data inside the SSD. In this paper, image retrieval is used as a showcase. As shown in Fig. 3, Cognitive SSD is designed to support the major components in the framework of DLG, allowing developers to customize and implement data retrieval solutions. Such a near-data retrieval system consists of two major components: the DLG library running in lightweight server that manages user requests, and the Cognitive SSD is plugged into the host server via the PCIe interface. As shown in Table 1, as the interface of Cognitive SSD system, the DLG library is established by leveraging the *Vendor Specific Commands* in the I/O command set of the *NVMe* protocol. It contains a *configuration library* and a *user library*. The configuration library enables the administrator to choose and deploy different deep learning models on the Cognitive SSD quickly according to application demand. After the feature-extracting deep hashing model has been deployed on the Cognitive SSD, a data processing request arriving at the host server could send and establish a query session to it by invoking the APIs provided by the user library. Then, the runtime system on the embedded processor of the Cognitive SSD receives and parses the request to activate the corresponding DLG-x module, which is associated with the user-created session. Next, we elaborate on the software and hardware design details of Cognitive SSD.

3.1 The Cognitive SSD Software: DLG Library

3.1.1 Configuration Library

Update Deep Learning Models: Because the choice of deep learning models significantly impacts the data retrieval system performance, the system administrator must be able to customize a specific deep hashing model according to the complexity and volume of database, and the quality of service measured by response latency or request throughput. Thereby, the configuration library provides a DLG-x compiler compatible with popular deep learning frameworks (i.e., Caffe) to allow the administrator to train the new deep learning model and generate corresponding DLG-x instructions offline. Then, the administrator can update the learning model running on the Cognitive SSD by updating the DLG-x instructions. The updated instructions are sent to the instruction area allocated in the NAND flash and stay there until a model change command (*DLG_config* in Fig. 3) is issued. Meanwhile, the DLG-x compiler also reorganizes the data layout of the DLG algorithm to fully utilize the internal flash bandwidth according to the structures of neural network model and graph, before the parameters of deep learning model and graphs are written to the NAND flash. The physical address of weight and graph structure information is recorded in the DLG-x instruction. In this manner, the DLG-x obtains the physical address of required data directly at runtime, instead of adopting the address translation or look-up operations that incur additional overhead. More details about the data reorganization scheme are introduced in § 4.

3.1.2 User Library

Data Plane: The data plane provides *SSD_read* and *SSD_write* APIs for users to control data transmission between the host server and the Cognitive SSD. These two commands operate directly on the physical address bypassing the flash translation layer. Users can invoke these APIs to inject data sent from users to the data cache region or the NAND flash on the Cognitive SSD based on the parameter of data address and data size. Afterwards, users can use those addresses to direct the operands in other APIs.

Task Plane: To improve the scalability of the DLG-x accelerator that supports deep hashing neural networks and graph search algorithms, we abstract the function of the DLG-x into three APIs in the task plane of user library: *DLG_hashing*, *DLG_index*, and *DLG_analysis*. These APIs are established using the ***C0h***, ***C1h***, and ***C2h*** commands of NVMe I/O protocol, respectively. All of them possess two basic parameters carried by NVMe protocol ***DWord***s: the *data address* indicating the data location in Cognitive SSD, and the *data size* in bytes.

First, the *DLG_hashing* API is designed to extract the condensed feature of input data and map it into the hash or seman-

Table 1: **DLG Library APIs for Cognitive SSD**

	-	API	NVMe command	DWord10	DWord11	DWord12	Description
Configuration Library	-	*DLG_config*	0xC3	address	size	Instruction/model	Update instruction and model on DLG-x
User Library	Task Plane	*DLG_hashing*	0xC0	data address	data size	Hashcode length	Extract the hashing feature of input data
		DLG_index	0xC1	data address	data size	T	Fast database indexing
		DLG_analysis	0xC2	data address	data size	User-defined	Analysis of input data
	Data Plane	*SSD_read*	0xC4	data address	data size	–	Physical Address Read
		SSD_write	0xC5	data address	data size	–	Physical Address Write

tic space, which is fundamental in a data retrieval system and useful for other analysis functions like image classification or categorization. This command contains an extended parameter: *hashcode length*, which determines the capacity of the carried information. For example, compared to the database with 500 objects types, the database with 1000 objects needs a longer hash code to avoid information loss. Second, the *DLG_index* API is abstracted from the graph search function of the DLG-x. It also includes an extended parameters: *T*, represents the number of search results configured by users based on the applications scenarios. Finally, the *DLG_analysis* API allows users to analyze the input data using the data analysis and processing ability of deep neural networks and it also possesses a reserved field for user-defined functions. These task APIs are the abstraction of the key near-data processing kernels provided by Cognitive SSD, and they can be invoked independently or combinedly to develop different in-SSD data processing functions. For instance, users could combine the *DLG_hashing* and *DLG_index* APIs to accomplish data retrieval on a large-scale database, where *DLG_hashing* maps the features of query data to a hash code and *DLG_index* uses it to search for the top-*T* similar instances.

3.1.3 Cognitive SSD Runtime The Cognitive SSD runtime deployed on the embedded processor inside the Cognitive SSD is responsible for managing the incoming extended I/O command via PCIe interface. It also converts the API-related commands into machine instructions for the DLG-x accelerator, as well as handles basic operations for NAND flash. It includes a request scheduler and the basic firmware. The request scheduler contains three modules: the ***DLG task scheduler***, the ***I/O scheduler***, and the ***DLG configurator***. The *DLG configurator* receives *DLG_config* commands from the host and updates the instructions generated by the compiler and parameters of the specified deep learning model for Cognitive SSD. The *DLG task scheduler* responds to users requests as supported in the task plane and initiates the corresponding task session in Cognitive SSD. The I/O scheduler dispatches I/O requests to the basic firmware or the DLG-x. The basic firmware includes the flash translation layer, for logical block mapping, garbage collection, bad block management functions, and communicates with the NAND flash controller for general I/O requests.

Note that the DLG-x accelerator occupies a noticeable portion of the flash bandwidth once activated, which perhaps degrades the performance of normal I/O requests. To alleviate this problem, instead of letting the task or I/O scheduler wait until the request is completed (denoted as Method A), the *DLG task scheduler* receives the NVMe command sent from host with doorbell mechanism and actively polls the completion status of the DLG-x periodically (denoted as Method B) to decide if the next request is dispatchable. We tested the normal read/write bandwidth of Cognitive SSD prototype described in § 5.1 with the Flexible IO Tester (fio) benchmark [4], under the worst-case influence where the DLG-x accelerator operations occupied all the Cognitive SSD channels. Experiments (Table 2) demonstrate that adopting Method B only causes a drop of 27%-44% in the normal I/O bandwidth whilst using Method A decreases almost 91% of the read/write bandwidth averagely when the DLG-x accelerator is busy dealing with the over-committed retrieval tasks.

Table 2: The I/O Bandwidth of Cognitive SSD.

-	I/O Bandwidth (MB/s) (I/O size = 128KB)			
-	Write	Random Write	Read	Random Read
Method-A	79.79	76.19	72.30	81.13
Method-B	524.86	421.23	654.31	698.98
Peak-Bandwidth	886.58	761.90	903.79	901.41

3.2 Hardware Architecture: Cognitive SSD

Fig. 3 depicts the hardware architecture of Cognitive SSD. It is composed of an embedded processor running the Cognitive SSD runtime, a DLG-x accelerator and NAND flash controllers connected to flash chips. Each NAND flash controller connects one channel of NAND flash module and uses an ECC engine for error correction. When the devices in each channel operate in lock-step and are accessed in parallel, the internal bandwidth surpasses the I/O interface. More importantly, though SSDs often have compact DRAM to cache data or metadata, the internal DRAM capacity can hardly satisfy the demand of the deep learning, which is notorious for its numerous neural network parameters. Worse still, the basic firmware like FTL and other components also occupy major memory resources. Therefore, the NAND flash controller is exposed to the DLG-x accelerator, which enables the DLG-x to read and write the related working data directly from NAND flash, bypassing the internal DRAM.

3.3 The Procedure of data retrieval in Cognitive SSD

Fig. 3 also depicts the overall process of Cognitive SSD when users perform unstructured data retrieval task. First, assume that the hardware instruction and parameters of Hash-AlexNet

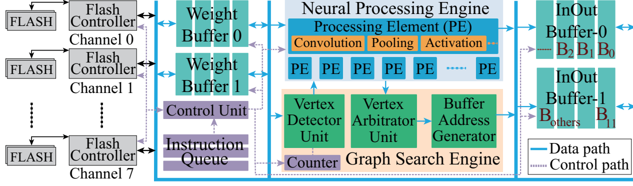


Fig. 4. The Architecture of DLG-x accelerator.

model have been generated and written to the corresponding region by leveraging the DLG-x compiler and the *DLG_config* command shown in Fig. 3. The Hash-AlexNet is the developer designated neural network for feature extraction of input data. Then, when the host DLG library captures a retrieval request, it packages and writes the user input data from the designated host memory space to Cognitive SSD through the *SSD_write* API. Meanwhile, the *DLG_hashing* command carrying the address of input data is sent to Cognitive SSD for hash code generation. Receiving the command, the request scheduler of the cognitive runtime parses it and notifies the DLG-x accelerator to start a hashing feature extraction session. Then, the DLG-x automatically fetches input query data from the command-specified data address and then loads deep learning parameters from NAND flash. Meanwhile, the other command, *DLG_index*, is sent and queued by the task scheduler. After the hash code is produced, the *DLG_index* is dispatched to invoke the graph search function in the DLG-x and uses the hash result to search the data graphs for relevant data entries. In this case, the DLG-x keeps fetching graph data from the NAND flash and sends the final retrieval results to the host memory once the task is finished.

4 DLG-x Accelerator

4.1 Architecture: Direct Flash Accessing

In contrast to a traditional hardware accelerator [20], the DLG-x accelerator is designed to directly obtain the majority of the working-set data from NAND flash. Fig. 4 illustrates the high-level diagram. The DLG-x accelerator has two activation buffers (InOut Buffer) and double-banked weight buffers. The intermediate results of each neural network layers are temporarily stored in the activation buffers, while the weight buffers act as a bridge buffer between the Neural Processing Engine (NPE) and the flash, which stream out the large quantity of neural parameters to the NPE. The NPE comprises a set of processing engines (PEs), which can perform fixed-point convolutions, pooling, and activation function operations. The Graph Search Engine (GSE) cooperates with the NPE and is responsible for graph search with the hash code generated by NPE. Both the NPE and GSE are managed by the control unit that fetches instructions from memory. Considering the I/O operation granularity of NAND flash, we reorganize the data layout of neural networks including both the static parameters and the intermediate feature data, to exploit the high internal flash bandwidth.

4.2 I/O Path in Cognitive SSD

Bandwidth Analysis: At first, we analyze and prove that the internal bandwidth of NAND flash can satisfy the demand of deep neural network running on the DLG-x accelerator. Assuming that the DLG-x and flash controller runs on the same frequency and the NPE unit of the DLG-x comprises N_{PE} PEs. The single channel bandwidth of NAND flash is BW_{flash}. Thereby, the bandwidth of M channels equals to:

$$BW_{flash}^{m} = M \times BW_{flash} \tag{1}$$

Suppose that a convolution layer convolves a $I_c \times I_h \times I_w$ input feature map (IF) with a $K_c \times K_h \times K_w$ convolution kernel to produce a $O_c \times O_h \times O_w$ output feature map (OF). The subscript c, h, and w correspond to the channel, height, and width respectively. The input/weight data uses an 8-bit fixed-point representation. It is easy to derive that the computation latency $L_{compute}$ and the data access latency L_{data} from NAND flash to produce one channel of feature map are:

$$L_{compute} = \frac{OP_{compute}}{OP_{PE}} = \frac{2 \times K_c \times K_h \times K_w \times O_h \times O_w}{2 \times N_{PE}} \tag{2}$$

$$L_{data} = \frac{S_{param}}{BW_{flash}^{m}} = \frac{K_c \times K_h \times K_w}{BW_{flash}^{m}} \tag{3}$$

Where the $OP_{compute}$ and S_{param} is the operation number and the parameters volume of a convolutional layer. OP_{PE} gauges the performance of the DLG-x measured in operations/cycle. To avoid NPE stalls,we must have $L_{compute} >= L_{data}$, and O_w is usually equal to O_h, so we have

$$O_w >= \sqrt{N_{PE}/BW_{flash}^{m}} \tag{4}$$

The above equation indicates that if only the width and height of the output feature map is larger than or equal to the right side of formula 4, which is four in our prototype with $N_{PE} = 256$ and $BW_{flash}^{m} = 16 bytes/cycle$, the NPE will not stall. For example, in the Hash-AlexNet mentioned in § 2.1, the minimum width of the output feature map in convolution layers is 7, which already satisfies in inequality 4 design. However, in the FC layers, $L_{compute}$ is smaller than L_{data}, so the data transfer time becomes the bottleneck. Thereby, the DLG-x accelerator only uses a column of PEs to deal with a FC layer because our prototype hardware design only supports eight channels, which does not meet inequality 4 with $M = 128$ and consequently causes PE underutilization. Besides, the parameter-induced flash reads will be minimized if the size of the weight buffer meets the condition: $S_{buffer} >= Max(K_c \times K_h \times K_w)$. The parameters exceeding the size of weight buffer will be repetitively fetched from the flash. To further improve the performance, we utilize ping-pong weight buffers to overlap the data loading latency with computation.

Data Layout in flash devices: Owing to the bandwidth analysis on the base of multi-channels data transmission, we propose flash-aware data layout to fully exploit flash bandwidth with the advanced NAND flash command-*read page cache*

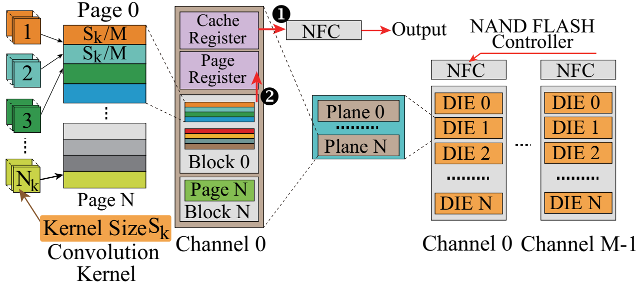


Fig. 5. The Data Layout in NAND flash.

command [11]. The read page cache sequential command provided by NAND flash manufacturer can continuously load the next page within a block into the data register (❷) while the previous page is being read to the buffer of the DLG-x or the cache region of the SSD from the cache register(❶). Thus, based on the NAND flash architecture with the provided page cache command, we choose to split the convolution kernels and store them into flash devices for parallel fetch. As shown in Fig. 5, assuming there are N_k convolution kernels with S_k kernel size, and M NAND flash channels are used by the DLG-x accelerator, each convolution kernel is divided into S_K/M sub-blocks and all such sub-blocks are interleaved to the flash channels. The convolution kernels exceeding the size of a page are placed into continuous address space in the NAND flash because the cache command reads out the next page automatically without any extra address or operation.

Data Flow: Taking the Hash-AlexNet as an example, when a request arrives at the DLG-x accelerator, the input data and the first kernel of the first convolution layer is transferred in parallel to the InOut buffer-0 and the weight buffer-0. After that, the DLG-x accelerator begins to compute the output feature map and stores them into the InOut buffer-1. When the first kernel is processed, the second kernel is being transferred from the NAND flash to weight buffer-1, then followed by the third and fourth kernel in sequence. Once the hash code is generated, it is sent to the graph search registers of NPE to locate the data structures similar to the query data if the *DLG task scheduler* decodes and dispatches a following *DLG_index* command.

4.3 Fusing Deep Learning and Graph Search

For fast and accurate database indexing, the DLG-x accelerator fuses the deep learning and graph search mechanism into unified hardware, and reuses the computation and memory resources for higher efficiency. Once the hash code of the query data has been generated, the DLG-x uses it to initially index the corresponding data graphs and searches for the closest data entries from graphs.

The graph search method originates from Navigating Spreading-out Graph [27] (NSG), which well fits the limited memory space of the Cognitive SSD for the large-scale multimedia data retrieval. The NSG algorithm includes an offline stage and an online stage. In the offline phase, the NSG method constructs a directed $K_{nbors}-NN$ graph for the storage data structures to be retrieved. In a graph, a vertex represents a data entry by keeping its ID and hash code. The unique ID represents a file and the hash code is the feature vector of this file, which could be obtained by invoking the *DLG_hashing* API in advance. The bit-width of ID (W_{id}) and hash code (W_{hash_code}) are user-configurable parameters in the API. In a graph, a vertex may be connected to many vertices, which have different distances from each other. However, only the top-K_{nbors} closest vertices of a vertex could be defined as its "neighbors", where K_{nbors} is also a reconfigurable parameter and enables users to pursue the trade-off between accuracy and retrieval speed. The DLG-x accelerator only accelerates the online retrieval stage and the database update occurs offline because the latter task is infrequent. The database update consists of hash code extraction stage and $K_{nbors}-NN$ graph construction stage, where the former is accelerated by the DLG-x accelerator and the latter is completed with the DLG library on CPUs. At offline graph construction, it takes about 10∼100 seconds to update the $K_{nbors}-NN$ graph on million-scale data on a server CPU. The hardware architecture for online graph search is presented in Fig. 4.

The graph search function of the DLG-x starts from evaluating the distance of random initial vertices in the graph and walks the whole graph from vertex to vertex in the neighborhood to find the closest results. As shown in Fig. 4, to maximize the utilization of on-chip memory, the weight buffer and InOut buffer are reused to store the neighbors of vertices and the search results of the graph search engine respectively. Since the Hamming distance (H-distance) is an integer value, the InOut buffer is divided into blocks according to the range of H-distance. For instance, the first block of the InOut buffer B_0 only stores the vertices with zero Hamming distance away from the query vertex, and the second block B_1 corresponds to the distance of one hop. The last area B_{others} stores the vertices with Hamming distance larger than the final value V_{final}, where the V_{final} is a re-definable parameter and calculated with formula 5.

$$V_{final} = \left\lfloor \frac{S_{buffer}}{D_{block} \times W_{id}} - 1 \right\rfloor \quad (5)$$

In the above equation, S_{buffer} is the on-chip buffer size of the DLG-x accelerator and D_{block} represents the number of vertex IDs that can be stored in each block. W_{id} is usually equal to 32bits. For instance, in our design, with $S_{buffer} = 256KB$ and $D_{block} = 5000$, it is easy to have $V_{final} = 12$. Note that the limited size of the region B_{others} cannot hold all the distant data vertices generated at runtime, and thus B_{others} is configured to a ring buffer to accommodate the incoming vertices cyclically.

A Vertex Detector Unit (VDU) is inserted to check whether the selected vertex has been evaluated. In VDU, the vertex will be discarded once found to have been walked before, otherwise it will be sent to the NPE unit to compute the Hamming distance from the query vertex. With the distance

provided by the NPE, the Buffer Address Generator (BAG) module allocates memory space in the InOut buffer for the vertex and then puts the vertex into the assigned areas of the InOut buffer. The unevaluated vertices will be fetched from the InOut buffer and the neighbors of these vertices are loaded from the weight buffer by the control unit. Meanwhile, the control unit will finally return the top-T closest vertices when the number of vertex in the InOut buffer reaches the threshold configured by users, where T is also configured by users via the *DLG_index* API.

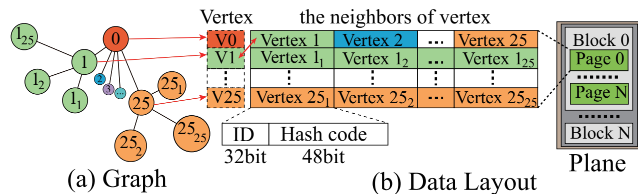


Fig. 6. The data organization on one page.

Data Layout for fast In-SSD NSG search: NAND flash read operations are performed at a page granularity (16KB), so that every time the DLG-x accesses the neighborhoods of one vertex (250bytes), it must read one whole page from flash, which perhaps causes low bandwidth utilization if locality is not well preserved. In our design that $K_{nbors} = 25$, $W_{id} = 32bits$ and $W_{hash_code} = 48bits$. Inspired by the intuition that the neighbor of a neighbor is also likely to be a neighbor of the query data in the graph, we can infer that the neighbors of the accessed vertex will be used soon due to the spatial locality. Therefore, as shown in Fig. 6, V_0 and all its neighbors $(V_1, V_2, ..., V_{25})$, are continuously aligned and stored from the beginning of a page. As a result, such a layout with redundancy is able to reduce flash access by 37x compared to a non-optimized graph layout. However, such a layout cause duplicates of vertices in storage and sacrifice additional storage space for better data access performance, which is worthwhile regarding the large capacity of SSDs.

Besides data layout transformation, the bit-width of the hash code is also worth elaborating. Due to the limited page size S_{page}, W_{hash_code} and K_{nbors} must conform to the resource constraint given by:

$$K_{nbors}^2 \times (W_{hash_code} + W_{id}) < S_{page} \quad (6)$$

Generally $S_{page} = 16KB$, and W_{id} is 32-bit wide and can represent 2^{32} files. Because the parameters W_{hash_code} and K_{nbors} directly impact the deep hashing performance by influencing the indexing accuracy and also the memory bandwidth consumption during graph search, once S_{page} is determined, W_{hash_code} and K_{nbors} must be adjusted to reach a perfect balance between accuracy and retrieval time at the offline stage. Thus, the DLG-x must support different parameter formats in order to achieve best-effort computing efficiency for databases of different volume and complexity.

Note that our graph layout and the according searching strategy are adapted to the underlying hardware for higher energy efficiency, and they will lead to a marginal amount

Table 3: The accuracy loss.

Dataset		Accuracy(%) at T samples				
		200	400	600	800	1000
CIFAR-10	Original	86.65	86.58	86.56	86.51	86.54
	Our	85.49	85.02	84.75	84.47	84.28
	Loss	1.16	1.56	1.81	2.04	2.26
ImageNet	Original	33.78	33.77	33.48	32.89	31.83
	Our	30.66	29.79	29.13	28.43	27.47
	Loss	3.12	3.98	4.35	4.45	4.36

of query accuracy losses compared to the original algorithm. Table 3 indicates the accuracy loss compared with the original lossless DLG algorithm (denoted as Original) on the CIFAR-10 [9] and ImageNet [45] datasets. The result shows that when *T=1000*, the accuracy drops by 2.26% and 4.36%, as a side-effect of the ~ 37x performance boost. Fortunately, the DLG library APIs are flexible enough to allow the developers to trade-off between accuracy and performance by manipulating the API arguments.

Data Flow: we show an example to brief the overall flow of the DLG-x based data retrieval. Firstly, when a query comes, the DLG-x fetches the input data and parameters of the deep learning model from the NAND flash into the InOut buffer and the weight buffer of the DLG-x respectively. Then, the NPE unit generates the hash code for the input data and writes it to the graph search registers of the NPE unit. After that, the DLG-x transfers the $K_{nbors} - NN$ graph from the NAND flash array to the weight buffer. At the first stage, the initial vertices are calculated and sent into the corresponding areas of the InOut buffer. At the second stage, the DLG-x control unit reads the first unevaluated vertex from the InOut buffer in ascending order of Hamming distance. Then, the graph search engine obtains the neighbors of the unevaluated vertex from the NAND flash and transfers the neighbors to NPE to generate their Hamming distances from the query vertex as well. Next, the Buffer Address Generator unit generates the write addresses for these neighbor vertices in the InOut buffers according to the calculated Hamming distance and writes these vertices to the InOut buffers. Meanwhile, the counter in the graph search engine determines whether the termination signal should be issued by monitoring the total number of vertices stored in the InOut buffer. Once the termination signal is generated, the Cognitive SSD runtime reads out the vertices from the InOut buffer and then transfers the ID-directed results stored in NAND flash to the host server via the PCIe interface.

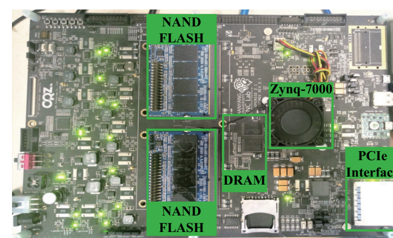


Fig. 7. Cognitive SSD prototype.

5 Evaluation

5.1 Hardware Implementation

To explore the advantages of the Cognitive SSD system, we implemented it on the Cosmos plus OpenSSD platform [7]. The Cosmos plus OpenSSD platform consists of an XC7Z045 FPGA chip, 1GB DRAM, an 8-way NAND flash interface, an Ethernet interface, and a PCIe Gen2 8-lane interface. A DLG-x accelerator is designed with DeepBurning [55] and integrated to the modified NAND flash controllers, and they are all implemented on the programmable logic of XC7Z045. The Cognitive SSD runs its firmware on a Dual 1GHz ARM Cortex-A9 core of XC7Z045. The Cognitive SSD is plugged into the host server via a PCIe link. The host server manages the high-level requests and maintains the DLG library for API calls. Fig. 7 shows the Cognitive SSD prototype constructed for this work.

5.2 Experimental Setup

We first selected the content-based image retrieval system (CBIR) based on deep hashing and graph search (DLG) algorithm as workload and evaluated the performance of DLG solution compared to other conventional solution (§5.3). we evaluated the DLG-x of the Cognitive SSD prototype in §5.4, and deployed the Cognitive SSD prototype to a single node and multi-node system, and evaluated them in §5.5, and §5.6, respectively. Except for the Cognitive SSD prototype, our experimental setup also consists of a baseline server running Ubuntu 14.04 with two Intel Xeon E5-2630 CPU@2.20GHz, 32GB DRAM memory, four 1TB PCIe SSDs and an NVIDIA GTX 1080Ti. Meanwhile, we implemented the CBIR system in C++ on the baseline server, where the deep hashing is built on top of Caffe [31]. Based on this platform, we constructed four solutions baselines: B-CPU, B-GPU, B-FPGA, and B-DLG-x. For B-CPU, the DLG algorithm runs on the CPU. For B-GPU, the deep hashing runs on the GPU and graph search runs on the CPU. For B-FPGA, we use ZC706 FPGA board [12] to replace Cognitive SSD, and the deep hashing runs on ZC706 FPGA board and graph search runs on the CPU. B-DLG-x implements the DLG algorithm on ZC706 FPGA board without any near-data processing technique compared to Cognitive SSD.

5.3 Evaluation of DLG algorithm

Experimental Setup. We used the precision at top T returned samples ($Precision@T$), measuring the proportionality of corrected retrieved data entries, to verify the performance of our deep hashing method on different models and datasets [36]. The performance is contrasted with traditional hash methods with 512-dimensional GIST feature, including Locality-Sensitive Hashing (LSH) [54] and Iterative Quantization (ITQ) [28]. The used datasets are listed in Table 4.

Evaluation. Fig. 8(a)-(d) shows the $Precision@T$ on different datasets with different deep hashing models. Due to the poor performance of LSH and ITQ [28] methods on ImageNet, we added the AlexNet-ITQ (A+ITQ) and AlexNet-CCA-ITQ (AC+ITQ) methods [58] that uses the output of the FC layer on Alexnet as the feature vector for search. Our DLG solution performs better than the other approaches on different datasets with different scales regardless of the choice of T value, especially compared to the conventional hash method. It also shows the robustness of the DLG solution when deploying on a real-world system. Meanwhile, Fig. 8(d) shows the performance of our approach is significantly improved when the code length increases to 64 bits. Thereby, the DLG-x accelerator is configured to support different hash code length to achieve the trade-off between retrieval accuracy and latency.

Table 4: Datasets used in our experiments

Dataset	Total	Train/Validate	Labels
CIFAR-10 [9]	60000	50000/10000	10
Caltech256 [29]	29780	26790/2990	256
SUN397 [57]	108754	98049/10705	397
ImageNet [45]	1331167	1281167/50000	1000

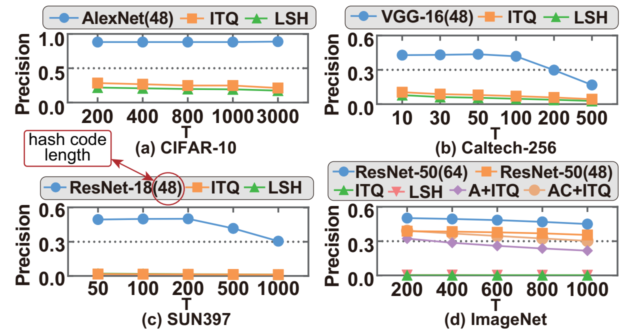


Fig. 8. Precision curves w.r.t top-T.

5.4 Evaluation of DLG-x

Experimental Setup. We implemented the deep hashing and graph search algorithm (DLG) on the DLG-x accelerator of Cognitive SSD and compared the latency and power of the DLG-x to the solutions based on CPU and GPU, where we ignore the FPGA baseline because its computational units are the same as the DLG-x. Firstly, we only compared the latency and power of the deep hashing unit on the DLG-x running various deep hashing models to CPU and GPU, where GPU only reports the total power consumed by NVIDIA GTX 1080Ti without the power of the CPU. Secondly, we only evaluated the latency of graph search function on the DLG-x with respect to different number (T) of top retrieved data entries on the CIFAR-10 and ImageNet dataset.

Performance. Firstly, Table 5 shows the latency of the DLG-x on various deep hashing schemes outperforms the solution based on CPU. While the latency of the DLG-x is higher than GPU because of the hardware resource and frequency limitation, it consumes less power compared to GPU. More importantly, the latency of the CPU and GPU on Table 5 only contains the computation delay of neural network without the delay of parameters transmission between storage and

Table 5: Deep hashing performance on different platforms.

Model	-	Latency (ms)	Power (Watt)
Hash-AlexNet	DLG-x	**38**	**9.1**
	CPU	114	186
	GPU	1.83	164
Hash-ResNet-18	DLG-x	**94**	**9.4**
	CPU	121	185
	GPU	7.13	112

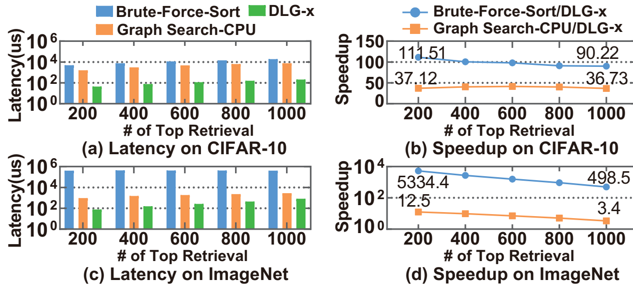


Fig. 9. Graph search performance of the DLG-x.

memory. When considering the delay of parameters transmission between storage and memory, the deep hashing occupies about 87.7% and 73.9% of the total processing time on the DLG-x accelerator and CPU baseline, on average, respectively. And the GPU only accounts for 3.5% of the total runtime on average because the high-speed data processing capability of GPU makes the data transmission becomes the bottleneck of system. Besides, the latency of deep hashing on the GPU occupies 54.17∼26.06% without considering the delay of data movement because the latency of graph search increases with the increase of T value.

Secondly, in this experiment, we utilized the Hash-AlexNet model to generate the hash code database for the construction of a $K_{nbors}-NN$ graph on the CIFAR-10 and ImageNet dataset. We compared the retrieval speed of the DLG-x accelerator with two counterparts: the brute-force search method that evaluates all the hash codes stored in the database, and the CPU executed graph search algorithm. The result is depicted in Fig. 9. For the CIFAR-10 dataset, the DLG-x accelerator is 111.51-90.22x and 37.12-36.73x faster than the brute-force method and the CPU-run graph search algorithms respectively, while for the ImageNet dataset it achieves a 5334.4-498.5x and 12.5-3.4x speed up over the latter two baselines. As introduced in § 4.3, the retrieval accuracy is only degraded by 2.26% and 4.36% when $T = 1000$ on the CIFAR-10 and ImageNet datasets, respectively.

Power Consumption. We measured and compared the power consumption of Cognitive SSD system with four baselines: B-CPU, B-GPU, B-FPGA, and B-DLG-x by using a power meter under two different situations: (1) IDLE: No retrieval requests need to respond, and (2) ACTIVE: A user continuously accesses the Cognitive SSD system. The result is illustrated in Table 6. When the Cognitive SSD+CPU system is IDLE, its power consumption is slightly higher than B-CPU and B-GPU because the Cognitive SSD prototype board IDLE

Table 6: Power consumption.

Power (Watt)	Cognitive SSD	Cognitive SSD+CPU	B-DLG-x	B-FPGA	B-CPU	B-GPU
IDLE	17	98.5	89	89	80	90
ACTIVE	20	122	185.7	195.6	186	330

Table 7: The hardware utilization of Cognitive SSD.

Module	#	LUT	FF	BRAM	DSP
Flash Controller	8	11031	7539	21	0
NVMe Interface	1	8586	11455	28	0
DLG-x Accelerator	1	67774	18144	137	197
`In Total`	1	203099	145078	354	197
Percent(%)	-	92.91	33.18	64.95	21.8

power is higher than that of the PCIe SSD and GPU. For active power, when delivering comparable data retrieval performance, the Cognitive SSD system reduces the total power consumption by up to 34.4% and 63.0% compared with B-CPU and B-GPU. Simply replacing the GPU with the FPGA board reduces power consumption by 40.7%. Furthermore, putting the DLG-x on an identical FPGA board without the NDP decrease power consumption by 43.72%, which is attributed to the efficiency of hardware specialization. Placing the DLG-x into the Cognitive SSD system further eliminates power consumption by another 19.3%, which is the benefit of near-data processing. In the case of the Cognitive SSD+CPU solution, the power of CPU is low because it is only responsible for instruction dispatch without any data transfer between storage and CPU. In other cases, the CPU is not only in charge of data transfer management but also for instruction dispatch or executing the graph search algorithm.

FPGA Resource Utilization. The placement and routing were completed with Vivado 2016.2 [8]. Table 7 shows the hardware utilization of Cognitive SSD. It only reports the resources overhead of the flash controller, NVMe controller, and the DLG-x accelerator module. The item of `In Total` counts in all FPGA resources spent by the Cognitive SSD.

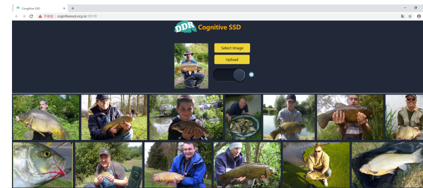


Fig. 10. A CBIR system based on Cognitive SSD.

5.5 The Single-node System Based on Cognitive SSD

Experimental Setup. We implemented the CBIR system by using the ImageNet dataset on the Cognitive SSD with a baseline server, where the baseline server is only responsible for receiving and sending retrieval requests to Cognitive SSD. The deep hashing architecture is a Hash-AlexNet network and the hash code length is 48 bits. As shown in Fig. 10, we built a web-accessible CBIR system based on web framework CROW [30] to evaluate the latency and query per second (QPS) of the system by simulating the user requests sent to

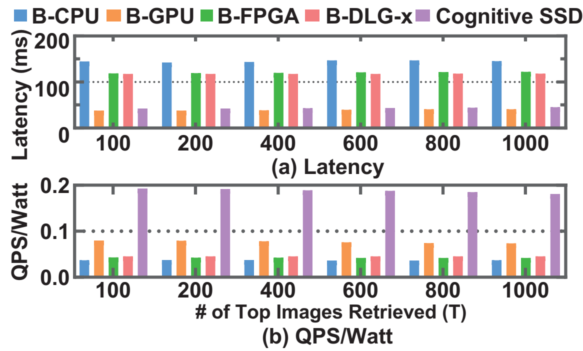


Fig. 11. System performance.

the URL address via ApacheBench (ab) [1]. The latency measurement indicates the time between issuing a request and the arrival of the result. The QPS is a scalability measuring metric characterizing the throughput of the system. The latency and QPS are affected by the software algorithm and the hardware performance of the system. Meanwhile, we utilized the metric of QPS per watt (QPS/Watt) to evaluate the energy-efficiency of the system.

Evaluation. We evaluated the performance of the Cognitive SSD system and four baselines under the assumption that data (weight/graph) cannot be accommodated in DRAM and must travel across the SSD cache, I/O interface, and DRAM before reaching a compute unit. The performance of the Cognitive SSD system and four baselines are shown in Fig. 11. With the increased number of top images retrieved, the retrieval time spent on the DLG-x accelerator will rise. It leads to increased retrieval latency and decreased QPS for the Cognitive SSD. Meanwhile, we also observe the 95% requests complete in time in experiments when write operations and garbage collection are inactive on the Cognitive SSD. Note that write operations and garbage collection are rare for the Cognitive SSD compared to read operations and usually occur offline. The workloads on the Cognitive SSD are read-only, which sustains the latency of the Cognitive SSD at a steady level with little fluctuation. Besides, in comparison to the B-CPU, the Cognitive SSD reduces latency by 69.9% on average. The performance improvement stems from the high-speed of data processing on the DLG-x accelerator compared to B-CPU. Due to the overhead of data movement caused by the bandwidth limitation of the I/O interface and onboard memory, the latency of B-FPGA and B-DLG-x is higher than B-GPU. Compared to the B-FPGA and B-DLG-x baselines, the Cognitive SSD reduces latency by 63.79% and 63.02% on average, which benefits from near-data processing. The average retrieval speed of B-GPU is 1.11x faster than Cognitive SSD because the execution of deep hashing costs more time on the resource-limited DLG-x compared to powerful GPUs. However, Cognitive SSD is more energy-efficient (QPS/Watt) than a GPU-integrated system by 2.44x, which is shown in Fig. 11(b). More importantly, the Cognitive SSD is implemented with FPGA and the operating frequency is only 100MHz. The performance will be better if the Cognitive SSD

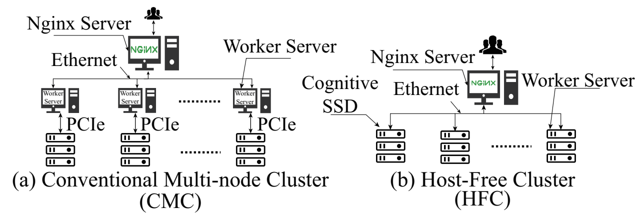


Fig. 12. The architecture of the conventional multi-node cluster(CMC)(a) and host-free cluster(HFC)(b).

is implemented with ASIC or escalated operating frequency.

5.6 The Cluster of Connected Cognitive SSDs

Experimental Setup. We evaluated the performance and the scalability of the Cognitive SSD when it scales into a multi-node cluster system. Fig. 12(a) shows the architecture of a conventional low-cost small-scale cluster system in a data warehouse. The cluster system consists of 10 worker servers and 1 Nginx [5] server. The Nginx server is responsible for load balancing. The worker nodes connect to the Nginx server by using TCP connections. In this case, we constructed four cluster system baselines by extending above four baselines: **BC-CPU**, **BC-GPU**, **BC-FPGA**, and **BC-DLG-x**. We issued requests to measure the QPS and the latency per request of the Cognitive SSD based cluster system when multiple users are accessing the web service shown in Fig. 10 concurrently via the ab tool.

Evaluation. Fig. 13(a) illustrates that when concurrent users equal to 400, all evaluated schemes rise slowly in experiments, which is limited by the thread of worker server and the node number of clusters. Meanwhile, the variation of peak QPS is due to the change of the performance bottleneck in different solutions. For example, the saturation performance of BC-GPU and CMC is constrained by the thread of the server while that of BC-CPU, BC-FPGA, and BC-DLG-x is determined by the latency of deep hashing inference and data movement. Fig. 13(b) shows that the QPS and latency per request of four baselines and CMC also change with the scale of the node cluster when the number of concurrent users reaches 400. As the number of nodes increases, the QPS gradually increases to the peak value, and the latency gradually decreases owing to improvement of service parallelism. When the nodes increase, the load-balancing mechanism of Nginx prevents a large hotspot formation in the cluster, which greatly increases the waiting time of requests.

We also measured the power consumption of the CMC system while running the CBIR service and compared it to BC-CPU, BC-GPU, BC-FPGA, and BC-DLG-x. When the cluster system is active, as shown in Fig. 14, the power consumption of a single node in the BC-CPU and BC-GPU are respectively 1.52x and 2.70x higher than a single node in the CMC. Similarly, Fig. 15 indicates the power of BC-CPU and BC-GPU is 1.45x and 2.46x than that of a CMC. Meanwhile, we also compared the QPS/Watt of CMC with other four baseline in Fig. 13(c). The energy-efficiency (QPS/Watt) of CMC

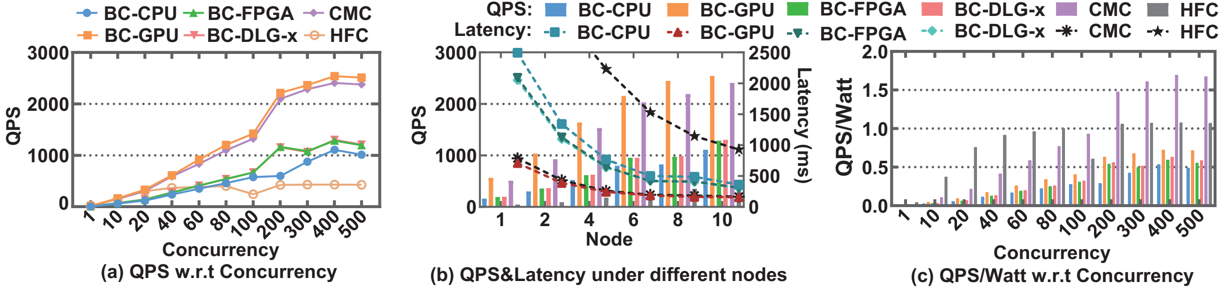


Fig. 13. Performance comparison.

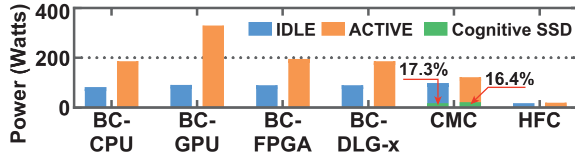


Fig. 14. Power dissipation of a single node.

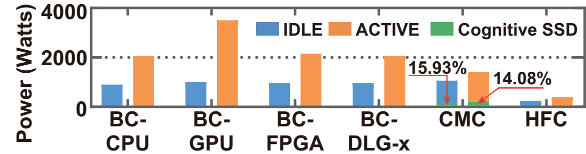


Fig. 15. Power dissipation of cluster.

is higher than the other four baselines because of short data movement path and the energy-efficient DLG-x accelerator. When the cluster system is IDLE, the power consumption is slightly higher than BC-CPU and BC-GPU because of the power consumption of the Cognitive SSD prototype is higher than the enterprise SSD.

It is noted that Fig. 15 indicates the power consumption of the Cognitive SSD only occupies about 15.93% (IDLE) and 14.08% (ACTIVE) of the entire system in the CMC architecture. The Cognitive SSD contains one Dual 1GHz ARM Cortex-A9 core, which could run embedded Linux system and has lower power consumption compared with the Intel Xeon CPU. Thereby, as shown in Fig. 12(b), to further reduce power consumption, we proposed the architecture of the host-free cluster (HFC) system, where the Cognitive SSD is directly connected to the Ngnix server via TCP connection, and the embedded Linux system runs a simple NAND flash management daemon and crow web framework.

We measured the performance of the host-free cluster system under the same experimental setup, which is illustrated in Fig. 13, Fig. 14, and Fig. 15. Fig. 14 shows that the power dissipation of a single node in the host-free cluster system is reduced by up to 89.2%, 93.9%, and 83.6% compared with that of BC-CPU, BC-GPU, and the original CMC when system is active. Considering the host server contains two Intel Xeon E5-2630 CPU that outperforms the dual Cortex-A9 in the Cognitive SSD, thereby, we measured the QPS per watt (QPS/Watt) to illustrate the energy-efficiency of the HFC system. The result (Fig. 13(c)) shows that when system concurrency is low, HFC delivers better energy-efficiency than the other four baselines and even better than the CMC architecture. The reason is that using high-performance machines to handle infrequent requests results in low energy-efficiency. Therefore, Fig. 13(c) witnesses the energy-efficiency of HFC relatively decreases with the increasing concurrency of the system. The performance growth of HFC under different node numbers also project that the level-off throughput is limited by the embedded CPU power instead of the DLG-x. In analysis, the HFC system will have much lower power consumption and higher performance if the Cortex-A9 processor is replaced by the latest Cortex-A series, e.g., a quad-core or octo-core Cortex-A75. Therefore, connecting the Cognitive SSD directly via interconnects contributes to much higher energy efficiency in Cognitive SSD system and guaranteed service throughput as well.

6 Conclusion

In this paper, we have introduced the Cognitive SSD, a near-data deep learning device that actively performs low latency, low power and high accuracy unstructured data retrieval. We have designed and implemented the Cognitive SSD with a direct flash-access deep hashing and graph search accelerator, to combat the complex software stack and inefficient memory hierarchy barriers in the conventional multimedia data retrieval systems. Our prototype demonstrates that the Cognitive SSD reduces latency by 69.9% on average compared to CPU, and more than 34.4% and 63.0% power saving against CPU and GPU respectively. Furthermore, the Cognitive SSD can scale to a multi-SSD system and significantly reduces the cost and power overhead of large-scale storage nodes in data centers. The demo of the retrieval system based on Cognitive SSD is available at [3] and part of the source code is available at [2].

Acknowledgments

We thank our shepherd, Joseph Tucek, and the anonymous ATC reviewers for their valuable and constructive suggestions. We thank the professor Jiafeng Guo of the CAS key lab of network data science and techology for his supports and suggestions. This work was supported in part by the National Natural Science Foundation of China under Grant 61874124, Grant 61876173,Grant 61432017, Grant 61532017, Grant 61772300 and YESS hip program No.YESS2016qnrc001.

References

[1] ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4. http://httpd.apache.org/docs/2.4/programs/ab.html.

[2] The Cognitive SSD. https://github.com/Cognitive-SSD.

[3] The Cognitive SSD Platform. http://cognitivessd.vicp.io:10110/.

[4] Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html#moral-license.

[5] NGINX. https://www.nginx.com/.

[6] Nvm express. https://nvmexpress.org/.

[7] The OpenSSD Project. http://openssd.io.

[8] Vivado. https://www.xilinx.com/support/download.html.

[9] Learning multiple layers of features from tiny images. Technical report, 2009. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf.

[10] The biggest data challenges that you might not even know you have, May 2016. https://www.ibm.com/blogs/watson/2016/05/biggest-data-challenges-might-not-even-know/.

[11] Micron nand flash. page 239, 2017. https://www.micron.com/products/nand-flash.

[12] ZC706 Evaluation Board for the Zynq-7000 XC7z045 SoC User Guide (UG954). page 115, 2018. https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf.

[13] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1629575.1629577.

[14] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, July 2014. https://ieeexplore.ieee.org/document/6871738.

[15] Matias Bjørling, Javier González, 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. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[16] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012. https://ieeexplore.ieee.org/document/6232366.

[17] Deng Cai. A revisit of hashing algorithms for approximate nearest neighbor search. *CoRR*, abs/1612.07545, 2016. http://arxiv.org/abs/1612.07545.

[18] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society. https://doi.org/10.1109/MICRO.2010.33.

[19] Intel IT Center. Big data 101: Unstructured data analytics. page 4. https://www.intel.com/content/www/us/en/big-data/unstructured-data-analytics-paper.html.

[20] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM. http://doi.acm.org/10.1145/2541940.2541967.

[21] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press. https://doi.org/10.1109/ISCA.2016.40.

[22] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM. http://doi.acm.org/10.1145/2464996.2465003.

[23] Hyeokjun Choe, Seil Lee, Seongsik Park, Sei Joon Kim, Eui-Young Chung, and Sungroh Yoon. Near-data processing for machine learning. *CoRR*, abs/1610.02273, 2016. http://arxiv.org/abs/1610.02273.

[24] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 9–16, Washington, DC, USA, 2013. IEEE Computer Society. http://dx.doi.org/10.1109/FCCM.2013.46.

[25] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM. http://doi.acm.org/10.1145/2463676.2465295.

[26] Cong Fu and Deng Cai. EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *CoRR*, abs/1609.07228, 2016. http://arxiv.org/abs/1609.07228.

[27] Cong Fu, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with navigating spreading-out graphs. *CoRR*, abs/1707.00143, 2017. http://arxiv.org/abs/1707.00143.

[28] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(12):2916–2929, December 2013. https://doi.org/10.1109/TPAMI.2012.193.

[29] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 Object Category Dataset, March 2007. http://resolver.caltech.edu/CaltechAUTHORS:CNS-TR-2007-001.

[30] Jaeseung Ha. crow: Crow is very fast and easy to use C++ micro web framework, June 2018. https://github.com/ipkn/crow.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. http://doi.acm.org/10.1145/2647868.2654889.

[32] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 1–13, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2749469.2750412.

[33] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 411–424, Piscataway, NJ, USA, 2018. IEEE Press. https://doi.org/10.1109/ISCA.2018.00042.

[34] Yangwook Kang, Yang-Suk Kee, Ethan L. Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2013. ftp://ftp.cse.ucsc.edu/pub/darrell/kang-msst13.pdf.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. http://doi.acm.org/10.1145/3065386.

[36] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement (v1.0). *CoRR*, abs/1610.02455, 2016. http://arxiv.org/abs/1610.02455.

[37] Wu-Jun Li, Sheng Wang, and Wang-Cheng Kang. Feature learning based deep supervised hashing with pairwise labels. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1711–1717. AAAI Press, 2016. http://dl.acm.org/citation.cfm?id=3060832.3060860.

[38] K. Lin, H. Yang, J. Hsiao, and C. Chen. Deep learning of binary hash codes for fast image retrieval. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 27–35, June 2015. http://ieeexplore.ieee.org/document/7301269/.

[39] V. E. Liong, Jiwen Lu, Gang Wang, P. Moulin, and Jie Zhou. Deep hashing for compact binary codes learning. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2475–2483, June 2015. http://ieeexplore.ieee.org/document/7298862/.

[40] H. Liu, R. Wang, S. Shan, and X. Chen. Deep supervised hashing for fast image retrieval. In *2016 IEEE*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 2064–2072, June 2016. http://ieeexplore.ieee.org/document/7780596/.

[41] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*, pages 773–785, 2017. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[42] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 257–270, 2013. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou.

[43] Jian Ouyang, Shiding Lin, Zhenyu Hou, Peng Wang, Yong Wang, and Guangyu Sun. Active ssd design for energy-efficiency improvement of web-scale data analysis. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 286–291, Piscataway, NJ, USA, 2013. IEEE Press. http://dl.acm.org/citation.cfm?id=2648668.2648739.

[44] 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*, ASPLOS '14, pages 471–484, 2014. http://doi.acm.org/10.1145/2654822.2541959.

[45] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015. http://dx.doi.org/10.1007/s11263-015-0816-y.

[46] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association. http://dl.acm.org/citation.cfm?id=2685048.2685055.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. http://arxiv.org/abs/1409.1556.

[48] Yongseok Son, Nae Young Song, Hyuck Han, Hyeonsang Eom, and Heon Young Yeom. A user-level file system for fast storage devices. In *Proceedings of the 2014 International Conference on Cloud and Autonomic Computing*, ICCAC '14, pages 258–264, Washington, DC, USA, 2014. IEEE Computer Society. https://doi.org/10.1109/ICCAC.2014.14.

[49] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 123:1–123:6, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2897937.2897995.

[50] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association. http://dl.acm.org/citation.cfm?id=2591272.2591286.

[51] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. Reducing data movement costs using energy efficient, active computation on ssd. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association. http://dl.acm.org/citation.cfm?id=2387869.2387873.

[52] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. *SIGARCH Comput. Archit. News*, 44(3):53–65, June 2016. http://doi.acm.org/10.1145/3007787.3001143.

[53] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 4:1–4:7, 2016. http://doi.acm.org/10.1145/2933349.2933353.

[54] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for Similarity Search: A Survey. *arXiv:1408.2927 [cs]*, August 2014. http://arxiv.org/abs/1408.2927.

[55] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In

Proceedings of the 53rd Annual Design Automation Conference, DAC '16, pages 110:1–110:6, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2897937.2898003.

[56] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014. http://dx.doi.org/10.14778/2732967.2732972.

[57] Jianxiong Xiao, Krista A. Ehinger, James Hays, Antonio Torralba, and Aude Oliva. Sun database: Exploring a large collection of scene categories. *Int. J. Comput. Vision*, 119(1):3–22, August 2016. http://dx.doi.org/10.1007/s11263-014-0748-y.

[58] Huei-Fang Yang, Kevin Lin, and Chu-Song Chen. Supervised learning of semantics-preserving hash via deep convolutional neural networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(2):437–451, February 2018. https://doi.org/10.1109/TPAMI.2017.2666812.

[59] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 87–100, 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang.

[60] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 477–492, Berkeley, CA, USA, 2018. USENIX Association. http://dl.acm.org/citation.cfm?id=3291168.3291203.

[61] Liang Zheng, Yi Yang, and Qi Tian. SIFT meets CNN: A decade survey of instance retrieval. *CoRR*, abs/1608.01807, 2016. http://arxiv.org/abs/1608.01807.

SIMD-X: Programming and Processing of Graph Algorithms on GPUs

Hang Liu
University of Massachusetts Lowell

H. Howie Huang
The George Washington University

Abstract

With high computation power and memory bandwidth, graphics processing units (GPUs) lend themselves to accelerate data-intensive analytics, especially when such applications fit the single instruction multiple data (SIMD) model. However, graph algorithms such as breadth-first search and k-core, often fail to take full advantage of GPUs, due to irregularity in memory access and control flow. To address this challenge, we have developed SIMD-X, for programming and processing of single instruction multiple, complex, data on GPUs. Specifically, the new Active-Compute-Combine (ACC) model not only provides ease of programming to programmers, but more importantly creates opportunities for system-level optimizations. To this end, SIMD-X utilizes just-in-time task management which filters out inactive vertices at runtime and intelligently maps various tasks to different amount of GPU cores in pursuit of workload balancing. In addition, SIMD-X leverages push-pull based kernel fusion that, with the help of a new deadlock-free global barrier, reduces a large number of computation kernels to very few. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving 3×, 6×, 24×, 3× speedup over Gunrock, Galois, CuSha, and Ligra, respectively.

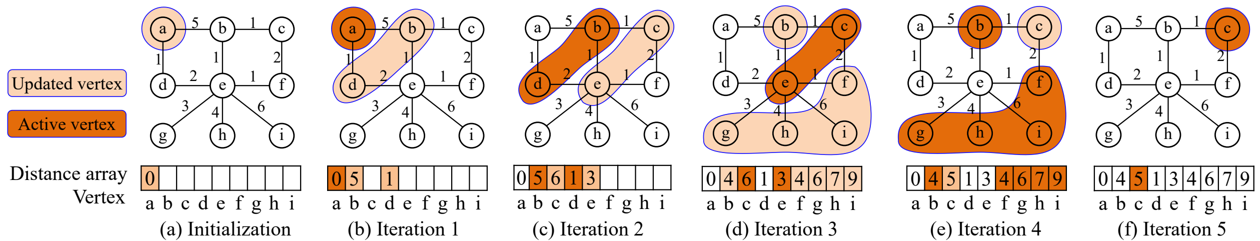
1 Introduction

The advent of big data [40, 27, 35, 36, 37, 5, 25, 26, 28, 14, 83] exacerbates the need of extracting useful knowledge within an acceptable time envelope. For performance acceleration, many applications utilize graphics processing units (GPUs) whose huge success comes from exploiting the data-level parallelism in these applications. Implicitly, the traditional single instruction multiple data (SIMD) model of GPUs assumes regular programming and processing, that is, not only the same instruction is executed but also the same amount of work is expected to perform on each piece of data. Unfortunately, neither assumption holds true for many emerging irregular applications, especially graph analytics which is the focus of this work. That is, such applications do not conform to the SIMD model, where different amount of work, or worse, completely different work, need to be performed on the data in parallel.

To enable graph computation on GPUs, this work advocates a new parallel framework, SIMD-X, for the programming and processing of *single instruction multiple, complex, data* on GPUs. At the heart of SIMD-X is the decoupling of programming and processing, that is, SIMD-X utilizes the *data-parallel* model for ease of expressing of graph applications, while enabling system-level optimizations at run time to deal with the *task-parallel* complexity on GPUs. With SIMD-X, a programmer simply needs to define what to do on which data, without worrying about the issues arisen from irregular memory access and control flow, both of which prevent GPUs from achieve massive parallelism.

SIMD-X consists of three major components: First, SIMD-X utilizes a new Active-Compute-Combine (ACC) programming model that asks a program to define three data-parallel functions: the condition for determining an *active* vertex, *computation* to be performed on an associated edge, and *combining* the updates from edge compute to vertex state. As we will show later, ACC is able to support a large variety of graph algorithms from breadth-first search, k-core, to belief propagation. While ACC adopts the Bulk Synchronous Parallel (BSP) model [49], it differs from traditional CPU-based graph abstractions such as edge- or vertex-centric models in that ACC avoids atomic operation, enables collaborative early termination (for BFS) and fine-grained task management on GPUs.

Second, SIMD-X relies on just-in-time (JIT) task management to balance parallel workloads across different GPU cores with minimal overhead. A good task list can increase not only parallelism, but also sequential memory access for the computation of next iteration, both



(a) Initialization (b) Iteration 1 (c) Iteration 2 (d) Iteration 3 (e) Iteration 4 (f) Iteration 5

Figure 1: SSSP on a graph, with nine vertices {a, b, c, d, e, f, g, h, i} and ten undirected edges (with weights). SSSP iteratively computes on the graph and generates the distance array. Particularly, heavy and light shadows represent active and most recently updated vertices, respectively.

of which are crucial for high-performance computing on GPUs. To this end, we have designed a set of new task management mechanisms, that is, online and ballot filters, each of which excels at the complementary scenarios, i.e., the former favors a small amount of tasks while the latter larger tasks. At runtime, SIMD-X judiciously selects the more suitable filter to assemble the active work list for the next iteration. Our JIT task management can largely reduce the memory consumption, thereby accommodate the graphs much larger than prior work [50, 77]. Moreover, SIMD-X delivers 16×, on average, speedup across various algorithms and graphs.

Third, SIMD-X designs a new technique of push-pull based kernel fusion which aims to further accelerate graph computing by reducing kernel invocation overhead and global memory traffic. SIMD-X addresses the deadlock issue which occurs in existing software global barrier [79] that is adopted by Gunrock [77]. Besides, instead of aggressively fusing the algorithm into one giant kernel, SIMD-X fuses the kernels around the pull and push stages within each computation to minimize both register consumption and kernel relaunching. The evaluation shows that the new fusion technique can reduce the register consumption by half and thus double the configurable thread count, leading to 42% and 25% performance improvement over non-fused and aggressive fusion, respectively.

SIMD-X is different from prior work in several aspects. First, despite an array of graph frameworks has surged, majority of them are for CPU systems while SIMD-X is for GPU accelerators that come with mounting programming challenges. In order to use GPUs efficiently, a programmer needs to possess an in-depth knowledge of GPU architecture [16, 1], e.g., Gunrock requires explicit management of GPU threads and memory [76], and B40C [50] and Enterprise [41] need thousands of lines of CUDA code for BFS specific optimizations. One of the goals of this work is to provide a simple programming model and delegate the responsibility of task management to SIMD-X. Second, current systems either ignore workload imbalance as in [33, 91], or resolve it reactively as in [76, 72], both of which result in undesired system performance. Lastly, because GPUs lack support for global synchronization, existing systems [73, 76, 41, 43, 69] either rely on the multi-kernel design or runtime tunning, both of which come with considerable overhead, especially for graph algorithms with high iteration count. SIMD-X addresses these challenges with new filters, and a deadlock-free software barrier.

2 SIMD-X Challenges and Architecture

2.1 Graph Computing on GPUs

Generally speaking, regular applications present uniform workload distribution across the data set. As a result, such applications lend themselves to the data-parallel GPU architecture. For development and evaluation, this work mainly uses NVIDIA GPUs, which have tens of streaming processors and in total thousands of Compute Unified Device Architecture (CUDA) cores [1, 56]. Typically, a *warp* of 32 threads execute the same instruction in parallel on consecutive data.

On the other hand, task management for irregular applications is challenging on GPUs. In this work, we focus on a number of graph algorithms such as breadth-first search, k-core, and belief propagation. Here we use one algorithm – Single Source Shortest Path (SSSP) – to illustrates the challenges. Simply put, a graph algorithm computes on a graph $G = (V, E, w)$, where V, E and w are the sets of vertices, edges, and edge weights. The computation updates the algorithmic metadata which are the states of vertices or edges in an iterative manner. A typical workflow of SSSP is shown in Figure 1. Initially, SSSP assigns the infinite distance to each vertex in the *distance array*, which is represented as blank in the figure. Assuming the source vertex is a, the algorithm assigns 0 as its initial distance, and now vertex a becomes active. Next, SSSP computes on this vertex, that is, calculating the updates for all the neighbors of vertex a. In this case, vertices $\{b, d\}$ have their distances updated to 5 and 1 in the distance array. At the next iteration, the vertices with newly updated distances become active and perform the same computation again. This process continues until no vertex gets updated. Different from breadth-first search, SSSP may update the distances of some vertices across multiple iterations, e.g., vertex b is updated in iteration 1 and 3.

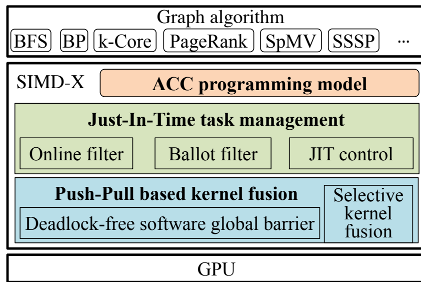


Figure 2: SIMD-X architecture

In this example, not every vertex is active at all time, and vertices with different degrees (number of edges) yield varying amounts of workloads. For instance, at iteration 3 of Figure 1(d), one thread working on vertex *c* computes two neighbors, while another thread on vertex *e* four neighbors.

2.2 Architecture

SIMD-X is motivated to achieve two goals simultaneously: providing ease of programming for a large variety of graph algorithms, whereas enabling fine-grained optimization of GPU resources at the runtime. Figure 2 presents an overview of SIMD-X architecture. To achieve the first goal, SIMD-X utilizes a simple yet powerful Active-Compute-Combine (ACC) model. This data-parallel API allows a programmer to implement graph algorithms with tens of lines of code (LOC). Prior work requires significant programming effort [50, 41, 76], or runs the risk of poor performance [33].

In SIMD-X, high-performance graph processing on GPUs is achieved through the development of two components: (1) JIT task management, which is responsible for translating data-parallel code to parallel tasks on GPUs. Essentially, SIMD-X "filters" the inactive tasks and groups similar ones to run on the underlying SIMD architecture. In particular, SIMD-X develops online and ballot filters for handling different types of tasks, and dynamically selects the better filter during the execution of the algorithm. And (2) Pull-push based kernel fusion. Graph applications are iterative in nature and thus require synchronizations. Fusing kernels across iterations would yield indispensable benefits, because kernel launching at each iteration incurs non-trivial overhead. In SIMD-X, we observe that with aggressive kernel fusion, register consumption would increase dramatically, lowering the occupancy and thus performance. To this end, SIMD-X deploys kernel fusion around pull and push stages of each graph computation, seeking a sweet spot that not only maximizes the range of each kernel fusion but also minimizes the register consumption. It is worthy noting that we also address the deadlock issue faced by software global barrier in SIMD-X.

3 ACC Programming Model

The novelty of SIMD-X lies at achieving both ease of programming and efficient workload scheduling, which is *especially hard on GPUs*. When it comes to graph computing, there are two main programming models: vertex-centric vs. edge-centric. Vertex-centric model, also referred to as "Think like a vertex" [49, 90] focuses on active vertices in a graph, whereas the latter one [61, 60] iterates on edges and simplifies programming.

3.1 Motivation

Graph programming converges to either *vertex-centric* or *edge-centric* models. In particular, the vertex-centric model contains two functions: `vertex_scatter` defines what operations should be done on this vertex, and `vertex_gather` applies the updates on the vertex. This model has been adopted by a number of existing projects, e.g., Pregel [49], GraphLab [45], PowerGraph [18], GraphChi [39], FlashGraph [90], Mosaic [47], and GridGraph [92], as well as GPU-based implementation such as CuSha [33] and Gunrock [76]. On the other hand, the edge-centric model is initially introduced by the external-memory graph engine X-stream [61] to improve IO performance. It requires a programmer to define two functions needed on each edge, `edge_scatter` and `edge_gather`. As such, this model schedules threads by the edge count. Particularly, one thread needs to send the information of the source vertex and the outbound edge to the destination vertex (`edge_scatter`), which atomically applies the new updates in `edge_gather`.

In this work, we believe the many-threaded nature of GPU architecture demands a new abstraction. We intend to exploit various thread scheduling options to better tackle workload imbalance [41, 77], while minimizing the overhead with regards to atomic operations on GPUs [46]. Table 1 summarizes the designs of recent GPU-based graph analytics systems. To avoid wasting the threads to compute on inactive vertices, *task filtering* is essential in generating a list of active vertices. Once task lists are ready, *workload imbalance* caused by skewed degree distribution in many graphs becomes the next concern. Since handling this issue in a vertex centric model involves nontrivial programming efforts [41], edge-based computing presents a desirable alternative. However, traditional edge-centric approach would result in atomic updates at the destination vertex, thus a proper schedule before applying the update is essential to *avoid atomic operation*. It is also important to note that *compressed sparse row* (CSR) is a preferable graph format which can save around 50% of the space over edge list format, as contemporary GPUs only feature tens of GB memory [1]. The proposed ACC framework is designed to address these three challenges.

Table 1: Comparison between ACC and relevant GPU-based programming models. [green] denotes desirable feature.

Abstraction	Related Work	Stages			Graph format
		Task filtering	Workload balancing	Avoid atomic operation	
ICU	CuSha [33], Lux [30]		Init/Compute (Edge)	Update	Edge list
ICRU	WS [32]		Init/Compute (Edge)	Reduce/IsUpdate	CSR
AFC	Gunrock [77]	Advance/Filter	Compute (Vertex, with atomic update)		CSR
GAS	GTS [34], GraphReduce [62]		Gather (Edge)	Apply/Scatter	Edge list
ACC	SIMD-X	Active	Compute (Edge)	Combine	CSR

3.2 ACC Model

The new ACC model contains three functions: **A**ctive, **C**ompute, and **C**ombine. ACC supports a wide range of graph algorithms and requires much fewer lines of code compared to prior work. In this following, we will discuss the three functions.

Active allows a programmer to specify the condition whether a vertex is active. Formally it can be defined:

$$\exists_v \leftarrow active(M_v, v)$$

where v is the vertex ID and M_v represents its metadata. Depending on the algorithm, the Active function may vary. Belief propagation (BP) is simple which treats all vertices as active. In comparison, SSSP, as shown in Figure 3(a), considers the vertices active when their current metadata differs from the prior iteration.

Simply put, SIMD-X distinguishes active vertices from inactive ones, and focuses on the calculation needed for each vertex. This is different from the vertex-centric model which deals with not only the active vertex but also its neighbors. Because two vertices may have different numbers of neighbors, existing systems [49, 18] likely suffer from workload imbalance. To this end, SIMD-X leverages a classification technique, similar to Enterprise [41], to group the active vertices depending on the expected workload.

Compute defines the computation that happens on each edge. In particular, it specifies the operations on the metadata of edge $(v,\ u)$ and two vertices v and u, which can be written as follows:

$$update_{v \to u} \leftarrow compute(M_v, M_{(v,u)}, M_u)$$

where the return value of $update_{v \to u}$ will be used by the Combine function. For example, SSSP can be defined as shown in Figure 3(a).

Combine merges all the updates, once the computations are completed. It can be represented:

$$update_u \leftarrow \bigoplus_{v \in Nbr[u]} update_{v \to u}$$

where $\oplus$ must be commutative and associative, e.g., sum and minimum, and is being applied to all the neighbors of vertex u. Figure 3(a) presents the Combine examples of SSSP. Particularly, BP summarizes all updates, where SSSP combines all updates from compute by selecting the minimum.

SIMD-X optimizes two types of combine operations, i.e., aggregation and voting. Particularly, aggregation cannot tolerate overwrites, that is, all updates are needed to arrive at the correct results. PageRank, SSSP and k-Core are representative examples of such operation. In contrast, voting relaxes this condition, that is, the algorithm is correct as long as one update is received because all updates are identical. For instance, BFS is valid once one parent vertex successfully visited the child vertex. Other algorithms, such as, weakly connected component and strongly connected component algorithms [67] also fall into this category.

3.3 Processing with ACC

This section uses SSSP an example to illustrate how the SIMD-X framework works. SSSP computes the shortest paths between the source vertex and the remaining vertices of the graph. Although similar to Breadth-First Search (BFS), SSSP is more challenging as only one vertex with the shortest distance should be computed at one time. To improve the parallelism, we adopt the delta-step [51] algorithm which permits us to simultaneously compute a collection of the vertices whose distances are relatively shorter. We assume positive edge weights.

As shown in line 12 - 21 of Figure 3(b), SIMD-X structures graph computation as a loop. Similar to popular GPU-based frameworks [77, 33, 32], ACC follows BSP model, that is, synchronization is required at the end of each iteration. As we will discuss in the next section, SIMD-X employ three kernels to balance the workload, Thread, Warp and CTA kernels working on small_list, med_list and large_list, respectively. During computing, the online_filter (Section 4) attempts to track the active vertices with the thread bins (i.e., small_bin, med_bin and large_bin). Note that each active vertex is stored in one of these three bins based upon its degree. After a deadlock free software global barrier (Section 5), SIMD-X checks whether an overflow happens in any of the thread bins, which leads to either a ballot filter-based active lists generation or a simple prefix-scan based concatenation of all thread bins to produce the active lists (line 17-21).

In Figure 3(b), Line 1 - 8 exemplifies the interactions between ACC and SIMD-X. Firstly, SIMD-X will schedule a warp of threads to work on the neighbors of one active vertex from med_list. Similarly, Thread and CTA will schedule a thread and CTA to work on each active vertex from small_list and large_list, respectively. During

```
1: Init (src){
2:     dist_curr [src] = 0;
3:     large_list.insert (src);
   }
4: Active (v){
5:     return dist_curr [v] != dist_prev [v];
   }
6: Compute (edge, weight){
7:     old_dist = dist_curr [edge.dest];
8:     new_dist = dist_curr [edge.src] + w;
9:     return old_dist > new_dist ? new_dist: old_dist;
   }
10: Combine (dist[]){
11:     return min (dist[]);
   }
                                        (a) SSSP in ACC
```

```
1: Warp (med_list, Compute, Combine, Active, overflow)
2:     for each active vertex v in med_list: //warp in parallel
           //Intra-warp parallel reduction.
           //Splitting compute and combine to avoid atomic operation.
3:         for each neighboring edge set edge[32] to vertex v:
4:             res [lane_id] = Compute ( edge[lane_id] );
5:         final = Combine (res[0 - 31]);
6:         if lane_id == 0:
7:             metadata_curr[v] = final;
8:             small_bin, med_bin, large_bin =
                       online_filter (Active, v, overflow);

   //Similar to Warp
   Thread(){//One thread working on one active vertex}
   CTA(){//One CTA working on one active vertex}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
10: SSSP_main{
11:     Init (src);
12:     while conditions:
            //Thread assignment (Workload balancing step II)
13:         Thread (small_list, Compute, Combine, Active, overflow);
14:         Warp (med_list, Compute, Combine, Active, overflow);
15:         CTA (large_list, Compute, Combine, Active, overflow);
16:         __software_global_barrier ();

            //Task management (Workload balancing step I)
17:         if (overflow):
18:             ballot_filer (small_list, med_list, large_list, Active);
19:         else:
20:             small_list, med_list, large_list = concatenate
                           (small_bin, med_bin, large_bin);
21:         __software_global_barrier ();
    }
                                        (b) ACC in SIMD-X
```

Figure 3: (a) SSSP in ACC model and (b) ACC abstraction and task management within SIMD-X framework.

computation, each thread will conduct a local Compute and Combine at line 4. Once finished, a cross Warp Combine happens at line 5. Eventually, the first thread from the Warp applies the final updates (without atomic operation) and store this vertex (if active) into corresponding thread bins.

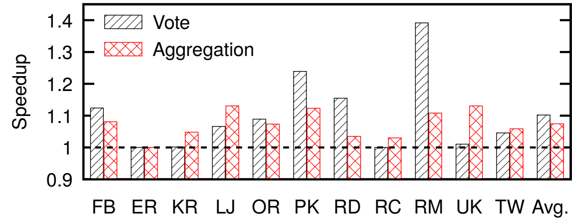


Figure 4: Speedup of our ACC model over Gunrock. Note vote and aggregation operations are materialized by BFS and SSSP algorithms, respectively, and x-axis contains the graph datasets which are defined in Table 3.

Comparison Figure 4 studies the performance impact of ACC vs. Gunrock. The new ACC model follows a computation then combine approach which pays the extra overhead (i.e., assembling all updates residing in shared memory from participating threads) in order to achieve the benefits of atomic-free updates. Gunrock, in contrast, directly applies the update to vertex status with atomic operations, thereby avoids inter-thread communication but experiences heavier overhead from atomic operation. One can see that ACC is, on average, 12% and 9% faster on vote and aggregation operations, respectively. For vote, the speedup comes from that ACC can schedule all threads to collaboratively determine early termination, which is not possible in Gunrock. Aggregation gains the performance from the elimination of atomic updates.

4 Just-In-Time Task Management

Workload balancing is essential for graph applications. The key is to ensure each GPU core, regardless of from which streaming processor, accounts for a similar amount of workload, which is often achieved with the following twin steps. Particularly, in **step I: task management**, the tasks are classified into various lists, namely small_list, med_list and large_list. In **step II: thread assignment**, various granularity of GPU threads are scheduled to work on different worklists. That is, a single thread per small task, a warp per medium task and a CTA per large task. Note, Figure 3(b) presents the pseudo code of step II and the bottom part of Figure 6 paints the corresponding workflow. We refer the readers to Enterprise [41] for more details regarding the landscape of this attempt.

Unlike prior work [41, 77, 50] which places particular efforts at step II, SIMD-X focuses on step I as we find it to be the major culprit that offsets the benefits of workload balancing. In the following, we will first analyze the drawback of existing batch filter method, then describe two new filters, and JIT selection mechanism.

Drawback of batch filter. This approach [76, 50, 11] first loads all the edges of the active vertices to construct an active edge list. Still using the example of SSSP in

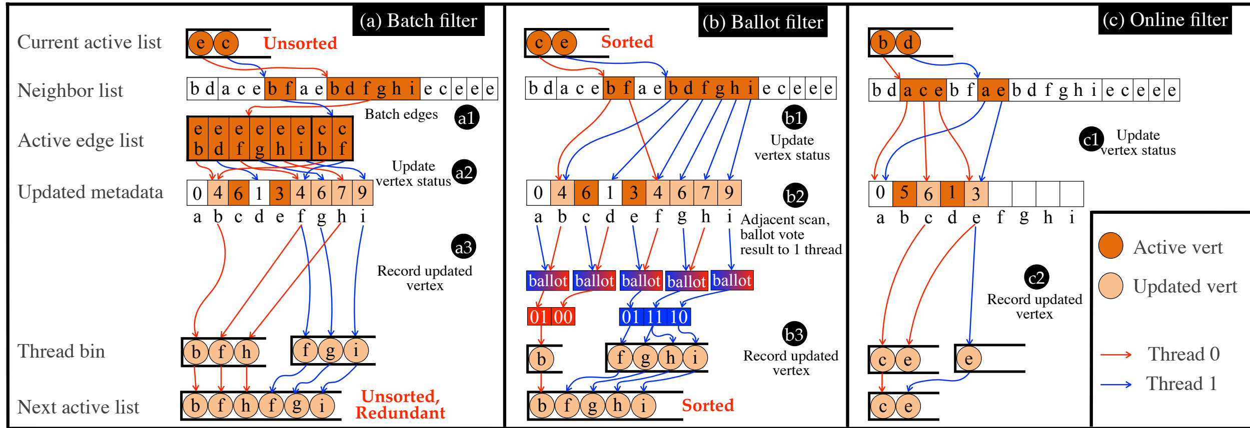


Figure 5: Three task management methods. Particularly, batch filter and ballot filter work on Figure 1(d) to produce a task list for next iteration. Online filter does that for Figure 1(c). Note, we assume the arrow flows of red and blue indicate the execution paths of red and blue threads.

Figure 1(c), this step loads neighbors of vertex {e, c} and constructs the active edge list in (a1) of Figure 5 (a). Next, batch filter checks these edges and updates vertex metadata (a2), followed by recording the updated vertices in thread bin at step (a3). Eventually, batch filter will concatenate these thread bins to arrive at a potentially *unsorted and redundant* next active list – $\{b, f, h, f, g, i\}$. Note, thread private local storage – thread bin – is used to avoid the expensive atomic operations, because multiple threads would need atomic operation to put active vertices directly into next active list.

We observe several drawbacks when using the batch filter for various graph algorithms. First, the active list can consume up to 2·|E| memory space because majority of the vertices in a graph can become active at one iteration [4, 41], which is especially true for popular social and web graphs. Considering GPU has very limited onboard memory (e.g., 16 GB), this restriction makes large-scale GPU-based graph computing intractable. Second, batch filter produces a worklist with unsorted and redundant active vertices, e.g., next active list – $\{b, f, h, f, g, i\}$ of Figure 5(a), which will lead to poor memory performance for next iteration computation.

Ballot filter is designed to overcome all these shortcomings. It first loads the neighbors of active vertices and immediately updates vertex metadata. As shown at step (b1) in Figure 5(b), the neighbors of {e, c} get updated immediately. Afterwards, thread 0 and 1 (red and blue lines) will exploit ballot scan to inspect the updated metadata and record those updated vertices in local thread bin at step (b3). The eventual step is similar to batch filter – we concatenate these two thread bins to get the next active list, whereas, with *sorted nonredundant* active vertices.

Ballot scan is the key to comprehend why we arrive at a better next active list. In steps (b2) and (b3) of Figure 5(b), threads 0 and 1 perform coalesced scan of vertex metadata, and with the CUDA *__ballot*() primitive, return a bit variable '01' to the first thread. Here 1 means active and 0 otherwise, in this case, vertex a is not active while b is. Through collaboratively working on the entire metadata array, the first thread eventually gets the bit value '0100' for the first four vertices, while the second thread '011110' for the remaining six vertices. Consequently, this approach produces a sorted active list, that is, $\{b, f, g, h, i\}$ in (b3).

We intentionally schedule thread 0 and 1 to collaboratively scan the metadata in order to achieve coalesced memory access during scan, as well as, making thread 0 and 1 account for a continuous range of vertices, that is, vertices a - d to thread 0 and e - i to thread 1. This achieves the dual benefits: coalesced scan and sorted active vertices in next active list. Last but not the least, this scheduling lends ballot filter to be many-thread safe.

We also notice an unpublished parallel efforts from Khorasani's dissertation [31] which is closely related to ballot filter. However, his design relies on atomic operation to compute the offsets of active vertices from each Warp in the next active list and subsequently assigns merely a single thread from the Warp to enqueue all these active vertices. This design implies twin disadvantages comparing to ours. First, atomic operation-based offset computation cannot yield sorted active lists. Second, single thread-based active vertices recording tends to be slower than Warp-based one which is our design.

Ballot filter is not without its own issue, especially when the amount of active vertices is low. In that case, scanning the metadata array would account for the majority of the runtime. For instance, in ER and RC graphs, 99.23% and 96.67% of the time is spent on scanning metadata in ballot filter alone solution, respectively.

Online filter is designed to accommodate the issue faced by ballot filter. In the first step, this method loads the ac-

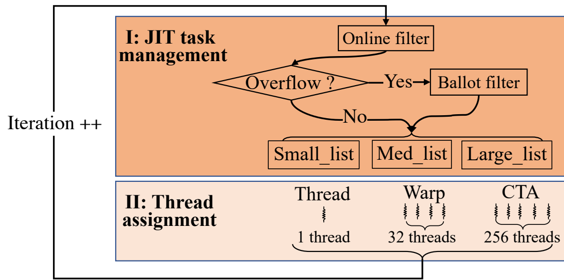


Figure 6: Workload balancing with the essential two steps: the novel JIT task management from SIMD-X and the thread assignment.

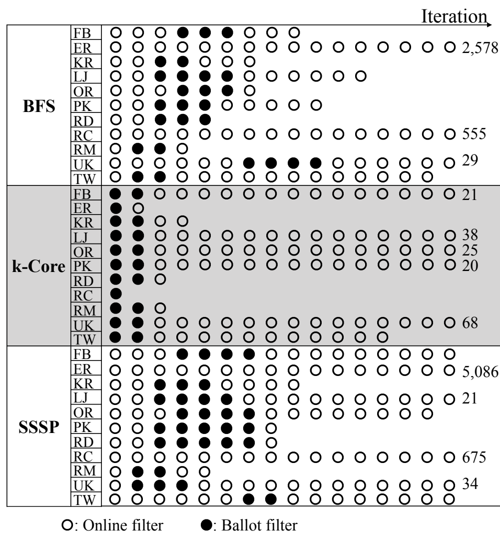
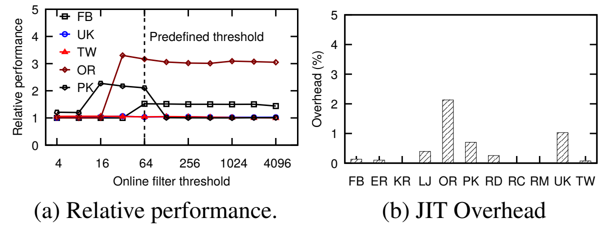


Figure 7: Ballot filter activation patterns.

tive neighbors, updates the destination vertex, and simultaneously records the active vertices in the thread bin. In the last step, it assembles all thread bins together as the next active list. When the number of active vertices is small, this approach turns out to be extremely fast. Here we use the early stage of SSSP as an example to explain its working process. As shown in Figure 5(c), $\{b, d\}$ are active vertices, this approach loads their neighbors for computation (c1), and immediately records the destination vertices. Eventually, it generates $\{e, c\}$ as the active list for the next iteration as shown in c2. It is also important to note that for online filter, the vertices in the active list may become redundant, and out of order.

In graph computing, it is possible that one GPU thread may encounter exceeding amount of active vertices, e.g., our tests on Twitter graph shows one GPU thread can reap more than 4,096 active vertices. Clearly, one cannot afford such a large thread bin for all threads, thus online filter will inevitably suffer from an overflow problem. Fortunately, ballot filter largely avoids this issue because it first updates the metadata of active vertices b2, which, to some extent, averages out the active vertices across threads in step b3.



(a) Relative performance. (b) JIT Overhead

Figure 8: The (a) relative performance of JIT management with respective to various online filter overflow thresholds on BFS and (b) the overhead of JIT on SSSP.

Just-In-Time control adaptively exploits ballot and online filters to retain the best performance. As shown in Figure 6, SIMD-X always activates the online filter first. Once a thread bin overflows, SIMD-X will switch on ballot filter to generate the correct task list for the next iteration. It is also worthy of mentioning that after JIT task management, we assign various granularity of threads to different lists in order to balance workload.

Interestingly, we find out that various algorithms and graph datasets present different selection patterns which tie closely to the amount of workload, that is, the higher volume of workload often results in the activation of ballot filter. As shown in Figure 7, BFS and SSSP typically use the ballot filter in the middle of the computation and online filter at the beginning and end. For high-diameter graphs, BFS and SSSP avoid the use of ballot filter. For instance, ER and RC always use the online filter along 2,578, 555, 5,086 and 675 iterations. k-Core activates the ballot filter at the initial iterations, i.e., typically the first two iterations except RC which only experiences one iteration because all its vertices have < 16 neighbors. At the extreme, BP and PageRank need the ballot filter at exactly the first iteration of computation.

Overflow thresholds for online filter. Clearly, this parameter directly determines when to switch on ballot filter, thereby affects the overall performance. Figure 8(a) presents the normalized performance with respect to various thresholds. As expected, a too low or too high threshold limits the performance because in either case, SIMD-X is forced to switch to ballot filter either too early or too late, leading to performance penalty. As such, in this work we select 64 as the predefined overflow threshold for all algorithms.

Overhead of online filter. After switching to ballot filter, JIT task management also executes the online filter in case it needs to switch back. Figure 8(b) studies the overhead of this design. On average, there is 0.02% slowdown, with the maximum of 2.1% observed for the OR graph. The reason for the small overhead is because online filter only tracks upto 64 (predefined threshold) active vertices for the next iteration and this operation is not on the critical path of the execution.

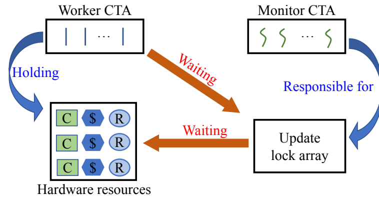


Figure 9: Deadlock in software global barrier, where ‘C’, ’$’, and ‘R’ represent core, L1 cache and register, respectively.

5 Push-Pull Based Kernel Fusion

Kernel fusion [73], a common optimization for a collection of iterative GPU applications, such as graph computing and deep learning [2, 58, 29, 10, 8], reduces expensive overhead of kernel invocation, as well as minimizes the global memory traffic as the life time of registers and shared memory is limited in each kernel. However, traditional efforts, such as Gunrock [77] and Xiao et al [79], fail to achieve cross the global barrier kernel fusion. This section starts with our observation and analysis of potential deadlock in the mainstream global barrier design [79, 82] and subsequently introduces a lightweighted deadlock free solution which enables the global thread synchronization within the fused kernel. However, aggressive kernel fusion requires a large amount of the registers and thus supports fewer parallel warps which could hurt the overall performance. To this end, we introduce a push-pull based kernel fusion to minimize the kernel invocation times and register consumption.

Software global barrier is needed to enable the balanced kernel fusion. Generally speaking, this approach uses an array – *lock* – to synchronize all GPU threads upon arrival and departure. During the processing, it assumes the thread CTA as the monitor while the remaining threads as workers. At arrival, each worker CTA updates its own *status* in *lock*. Once all worker CTAs have arrived, the monitor changes the *statuses* of all CTAs to *departure*, allowing all threads to proceed forward.

This approach, unfortunately, suffers from potential deadlock [79], as illustrated in Figure 9. Specifically, the worker thread CTAs may hold all GPU hardware resources, such as streaming processors, registers and shared memory, while waiting for the monitor to update the *lock* array. In the meantime, the monitor cannot update the *lock* array, due to lack of hardware resources (e.g., thread over subscription).

Compiler-based deadlock free barrier. SIMD-X utilizes the barrier in a way to ensure that every CTA, regardless of a work or the monitor, can obtain hardware resources when needed. This is achieved through comparing the resources needed by the kernels, against the total available resources. Based on the GPU architecture, we can obtain the total amount of registers (*#registerPerSMX*) that can be provided by each streaming processor, e.g., 65,536 registers of NVIDIA K40 GPUs and 32,768 from K20 GPUs. On the other hand, we can collect the register consumption (*#registerPerThread*) of each kernel at the compilation stage. Putting together, SIMD-X is able to calculate the appropriate thread configuration for kernels.

The number of CTA can be computed as follows:

$$\#CTA = floor(\frac{\#registersPerSMX}{\#registersPerThread \cdot \#threadsPerCTA}) \cdot \#SMX \quad (1)$$

where *#threadsPerCTA* is configured by a user, i.e., 128 by default. For example, when deploying a kernel, each thread consumes 110 registers, and on K40 that contains 15 SMXs, each of which contains 65,536 registers. If *#threadsPerCTA* is set to 128, one gets $\#CTA = ceil(\frac{65536}{110 \times 128}) \times 15 = 60$. As a result, we can configure this kernel as CTA and thread count per CTA as 60 and 128, respectively.

Notably, portable Inter-Block Barrier [69] is closely relevant to our effort. However, this method proposes extremely complicated thread block management mechanism that requires to distinguish whether one thread block will execute useful workloads or not during runtime. This requires nontrivial programmer efforts and scheduling overhead. In comparison, our method achieves this deadlock-free configuration before runtime and is completely transparent to the end users.

Push-Pull based kernel fusion. As shown in Table 2, the register consumption (using the compilation flag *-Xptxas -v*) increases from average 25 to 110, that is a 4.4× difference. Note, consuming too many registers will curb the number of active threads (according to equation 1). Unfortunately, majority of the graph algorithms are data intensive, thus prefer a higher volume of active threads because more active threads can better hide the frequent memory access stalls caused by data intensive applications. Consequently, we need a balanced fusion strategy that keeps both register consumption and kernel invocation low.

To this end, SIMD-X leverages the push-pull model used in the graph algorithms. That is, such algorithms often use push or pull based computing in several consecutive iterations. Lines 12 - 21 from Figure 3(b), for example, discuss the pull model and we can fuse these lines into a single GPU kernel. Similarly, push model can also be fused into a single kernel. Section 6 details how pull/push iterations occur in various graph algorithms.

SIMD-X adopts the pull-push model as in [66, 4, 41], by controlling where (in/out edge) Compute happens and how to Combine the results and apply (in atomic or atomic free manner). Particularly, in the push model, SIMD-X conducts Compute on the out neighbors of each active vertex, and relies on atomic operations to apply the

Table 2: Register consumption for various kernels.

Kernel	Push (no fusion)				Pull (no fusion)				Selective fusion		All fusion
	Thread	Warp	CTA	Task mgt	Thread	Warp	CTA	Task mgt	push	pull	
Register consumption	26	27	28	24	24	24	22	30	48	50	110
Kernel launching count	up to 40,688								3		1

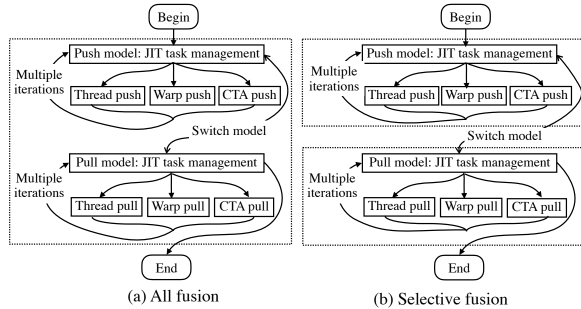


Figure 10: Consecutive iterations from graph algorithms often cluster to push and model computation separately: (a) all fusion, (b) selective fusion.

Table 3: Graph Dataset.

Graph Name	Abbrev.	Vertex Count	Edge Count
Facebook	FB	16,777,215	775,824,943
Europe-osm	ER	50,912,018	108,109,319
Kron24	KR	16,777,216	536,870,911
LiveJournal	LJ	4,847,571	136,950,781
Orkut	OR	3,072,626	234,370,165
Pokec	PK	1,632,803	61,245,127
Random	RD	4,000,000	511,999,999
RoadCA-net	RC	1,971,281	5,533,213
R-MAT	RM	3,999,983	511,999,999
UK-2002	UK	18,520,343	596,227,523
Twitter	TW	25,165,811	787,169,139

updates to the destination vertices. In contrast, the pull model schedules Compute on the in neighbors of active vertices, and uses atomic-free strategy to Combine all updates and apply to the destination vertices. As different iterations favor one model over the other, we follow a similar rule as in Ligra [66] to alternate between the push and pull models. That is, when the workload on the push model works on more than 30% of the edges, SIMD-X will switch to pull model.

The idea of push-pull based kernel fusion is to fuse kernels around the pull and push computing. In other words, for the push-based iterations, SIMD-X fuses different compute kernels (for thread, warp, CTA), as well as task management kernel, into one push kernel. The kernel only terminates when the computation finishes or it needs to switch to pull computing according to the criterion discussed in Section 3.3. Similar optimizations are done for the pull-based iterations.

Using the new push-pull based fusion, the register consumption decreases to 48 and 55 thus increases the configurable thread count by 50%. Table 2 presents the register consumption and kernel invocation of different kernel fusion techniques. By using the push-pull based kernel fusion, the kernel relaunch is merely three while its register consumption is cut by half.

6 Graph Algorithms and Datasets

In addition to SSSP that is discussed in Section 3.3, this section further presents a variety of algorithms which are implemented on SIMD-X to examine the expressiveness of ACC programming model, and performance impacts of task management and kernel fusion techniques.

BFS [41] traverses a graph level by level. At each level, it loads all neighbors that are connected to vertices visited in the preceding level, inspects their statuses (metadata), and subsequently marks those unvisited neighbors as active for the next iteration. Notably, BFS conducts synchronizations at the end of each level, relies on vote to combine the updates. During the entire process of traversal, BFS typically experiences light workload at the beginning and end of the computation while heavy workload in the middle.

Belief propagation (BP), also known as sum-product message passing algorithm, infers the posterior probability of each event based on the likelihoods and prior probabilities of all related events. Once modeled as a graph (Bayesian network or Markov random fields), each event becomes a vertex with all incoming vertices and edges as related events and corresponding likelihoods. In BP, vertex possibility is the metadata.

***k*-Core** (KC), which is widely used in graph visualization application [42, 53], iteratively deletes the vertices whose degree is less than k until all remaining vertices in this graph possess more than k neighbors. k-Core experiences large volume of workloads at initial iterations and follows with light workloads. This work uses a default value of $k = 16$.

PageRank (PR) [57] updates the rank value of one vertex based on the contribution of all in-neighbors iteratively till all vertices have stable rank values. Because the contributions of in neighbors are summarized to the destination vertex, we start PageRank with the pull model and *agg_sum* as the merge operation. At the end of PageRank, we switch to the push model because the majority of the vertices are stable [87]. The switch is decided by a decision tree.

Graph Benchmarks. We evaluate on a wide range of graphs as shown in Table 3, which falls into four types, i.e., social networks, road maps, hyperlink web and synthetic graphs. Particularly, Facebook [17], LiveJournal [68], Orkut [68], Pokec [68], and Twitter [38] are common social networks. Europe-osm [12] and

RoadCA-net [70] are two large roadmap graphs, and UK-2002 [70] is a web graph. Furthermore, we use Graph500 generator to generate Kron24 [6], and GTgraph [19] for R-MAT and random graphs. Europe-osm and RoadCA-net are high diameter graphs, with 2570 and 555 as their diameters, respectively. LiveJournal, Pokec, Twitter and UK-2002 are medium diameter graphs, i.e., 10 - 30 as their diameters. The diameters of the remaining graphs are all smaller than 10. For graphs without edge weight, we use a random generator to generate one weight for each edge similar to Gunrock [76]. These graphs are stored in *compressed sparse row* (CSR) format.

7 Experiments

We implement SIMD-X[1] with 5,660 lines of CUDA and C++ code. All the algorithms presented in Section 6 are implemented with around 100 lines of C++ code. The source code is compiled by GCC 4.8.5 and NVIDIA nvcc 7.5 with the optimization flag as O3. In this work, we evaluate SIMD-X on a Linux workstation with two Intel Xeon E5-2683 CPUs (14 physical cores with 28 hyperthreads), and 512GB main memory. Throughout the evaluation, we use uint32 as the vertex ID and uint64 as index and evaluate our system on NVIDIA K40 GPUs unless otherwise is specified. We also test SIMD-X on earlier K20 and latest P100 GPUs. The timing is started once the graph data is loaded in GPU global memory. Each result is reported with an average of 64 runs.

7.1 Comparison with State-of-the-art

Table 4 summarizes the runtime of SIMD-X against Galois and Gunrock which are state-of-the-art CPU and GPU graph processing systems, respectively, as well as CuSha (GPU) and Ligra (CPU), two popular graph frameworks. The take aways of this table are two folds.

First, SIMD-X is both space efficient and robust. As one can see, since CuSha requires edge list as the input for computation, it cannot accommodate large graphs (e.g., FB and TW) across all algorithms. Besides, since Gunrock requires large amount of space for batch filter, it suffers out of memory (OOM) error for all larger graphs in SSSP. Even CPU systems (Galois and Ligra) enjoys affluent memory space (512 GB) from CPU, they cannot converge to a result for high diameter graphs. That is, Galois cannot converge for SSSP on ER while Ligra fails to obtain result for BFS on UK graph.

Second, SIMD-X outperforms all graph processing frameworks. In general, SIMD-X is 24×, 2.9×, 6.5× and 3.3× faster than CuSha, Gunrock, Galois and Ligra, respectively. In BFS, SIMD-X bests CuSha, Gunrock, Galois and Ligra by 9.6×, 4.8×, 2.1× and 2.4×, respectively. We also notice that SIMD-X is slower than Galois on the RD graph because workload balancing brings negligible benefits to uniform-degree graph (RD). Also, SIMD-X is slightly worse than Ligra on RM graph since this graph only has a diameter of four thus both JIT task management and kernel fusion brings trivial benefits to GPU based graph systems, as evident by much lower performance on CuSha and Gunrock.

In PageRank, SIMD-X achieves 1.2×, 2.1×, 2.3× and 4× speedups over CuSha, Gunrock, Galois and Ligra, respectively. Note, even CuSha cannot support all large graphs due to large memory space consumption, it performs roughly similar to SIMD-X with even outperforming SIMD-X on LJ and OR. This is generally because PageRank tends to be more computation intensive than other graph algorithms and needs to compute all edges, curbing the benefits of task management and kernel fusion. However, edge list format (of CuSha) doubles memory consumption, facing OOM for large graphs.

For SSSP, SIMD-X wins 21×, on average, over all four projects. We project SIMD-X to better outperform all systems than observed for BFS algorithm because SSSP experiences more iterations with larger volume of active tasks, placing more favor towards ACC model, JIT task management and push-pull based kernel fusion. However, because Gunrock fails to accommodate all large graphs, our benefits cannot surface – ending with merely 1.8× speedup. Second, CuSha spends 519,674 ms on the high diameter ER graph which is 480× slower than SIMD-X because task management is absent from CuSha. We also notice Galois performs better than SIMD-X in RD, again, due to the uniform degree distribution.

For k-Core, where $k = 32$, SIMD-X wins Ligra by 20×. Such a striking advantage comes from three parts. First, as reflected by Figure 11(b), k-Core generates extensive amount of workload variations thus benefits tremendously from JIT task management. Second, k-Core's iterative nature also enjoys the benefits from push-pull based kernel fusion, as shown in Figure 12(c). Lastly, the flexibility of ACC allows innovative k-Core algorithm designs – we will stop further subtracting the degree of destination vertex once the destination vertex's degree goes below k – this reduces tremendous unnecessary updates. Note comparisons of Belief Propagation, as well as other systems for k-Core are not included because those systems fail to support such algorithms.

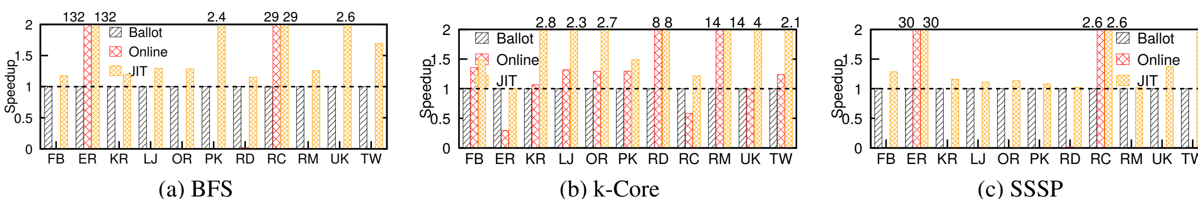
7.2 Benefits of Various Techniques

This section studies the performance impacts brought by JIT task management and push-pull based kernel fusion. As we have presented in Section 4, JIT task management only works for applications that experience work-

[1] SIMD-X source `https://github.com/asherliu/simd-x`.

Table 4: Runtime (ms) of SIMD-X and Gunrock, and Galois. A K40 GPU is used to test SIMD-X and Gunrock, and a CPU with 28 threads for Galois. The **blank space** indicates the test cannot complete for the given algorithm and graph.

Alg	System	FB	ER	KR	LJ	OR	PK	RD	RC	RM	UK	TW	Avg. speedup
BFS	SIMD-X	198	400	130	59	40	20	82	15	47	308	210	-
	CuSha			988	224	341	72	435	297	674	4298		9.6
	Gunrock	685	849	677	71	225	44	647	146	506	312	697	4.8
	Galois	482	1068	140	139	42	34	48	53	65	229	322	2.1
	Ligra	1086	1426	176	89	51	31	88	48	40		496	2.4
PR	SIMD-X	1553	346	1141	236	435	118	1105	13	800	637	1525	-
	CuSha			1704	182	323	180	1402	15	886			1.2
	Gunrock	3004	884	3129	275	927	166	2963	43	2208	784	3180	2.1
	Galois	4552	603	3069	424	1061	218	3576	20	2067	842	4178	2.3
	Ligra	16780	1368	2000	1324	1786	310	809	35	1703		9360	4
SSSP	SIMD-X	1816	1080	998	284	604	143	1505	223	478	703	1344	-
	CuSha		519674	1663	692	1120	260	1610	438	1236			62
	Gunrock		1206	1220	431	1259	336	5059	229				1.8
	Galois	161596		8485	1785	1166	356	747	3440	5877	9081	1818	15
	Ligra	14067	3043	2893	1627	1567	605	3353	301	2783	1300	5217	3.7
k-Core	SIMD-X	366	78	131	60	63	33	10	4	19	151	277	-
	Ligra	6337	1167	2813	1707	1700	654	27	36	235	6627	5783	20



(a) BFS (b) k-Core (c) SSSP

Figure 11: Benefit of just-in-time task management, normalized to the performance of the ballot filter.

load variations, that is, BFS, k-Core and SSSP. On the other hand, push-pull based kernel fusion is applicable for all five algorithms

On average, JIT task management presented in Figure 11, is 16×, 26× and 4.5× faster than the ballot filter for BFS, k-Core and SSSP. As expected, online filter alone cannot work for many graphs, particularly large ones, e.g., Facebook, Twitter and UK2002 graphs in BFS and SSSP. Without considering overflow problem (ER and RC graphs), JIT task management adds a small 1-2% overhead on top of the online filter on BFS and SSSP.

On k-Core, JIT task management is, on average, 28.5× and 5% faster than ballot and online filter, respectively. We also observe that the ballot filter outperforms the online filter on ER and RC graphs by 3.4× and 1.7×, because k-Core removes a large volume of vertices which favors the former to produce a non-redundant and sorted work list.

Push-pull based kernel fusion brings, on average, 43% and 25% improvement over non-fusion and all-fusion across all algorithms and graphs. In particular, push-pull based kernel fusion tops non-fusion by 74%, 11%, 85%, 10% and 66% on BFS, BP, k-Core, PageRank and SSSP. BFS, k-Core and SSSP achieves more performance gains because they are not computation intensive and tend to run a higher number of iterations. For all fusion, our new kernel fusion is 55%, 6%, 62%, 25% and 11% faster on BFS, BP, k-Core, PageRank and SSSP. It is important to note that all fusion is not always beneficial, i.e., all fuse option of PageRank is average 13% slower than no fusion because all fusion limits the amount of configurable threads. However, for memory intensive applications, like BFS and SSSP on ER and RC, all fusion is on average 2× better.

7.3 Performance on Different GPUs

We also evaluate SIMD-X, Gunrock and CuSha on various GPU models, such as K20 and P100 GPUs. It is not surprising to see tht SIMD-X presents the biggest performance gain on the latest GPUs. In detail, SIMD-X on K40 and P100 performs 1.7× and 5.1× better than K20 GPU. In contrast, Gunrock merely gets 1.1× and 1.7× performance improvement when moving from K20 to K40 and P100, respectively. Similarly for CuSha, its performance on K40 and P100 are 1.2× and 3.5× better than K20, respectively. The root cause of this disparity is that SIMD-X's kernel fusion technique can dynamically configure its GPU kernels to the fitting thread count on the corresponding hardware so as to achieve the peak performance. For instance, the thread count increases by 1.2× and 5.1× on K40 and P100 than on K20 for BFS.

8 Related Work

Recent advance in graph computing falls in algorithm innovation [51, 87, 15], framework developments [49, 18, 45, 39, 42, 90, 92, 22, 66, 63, 61, 23, 54, 60, 74, 7, 80, 84, 65, 88, 75, 55, 89, 86, 85, 3, 78, 52, 21, 9, 81] and accel-

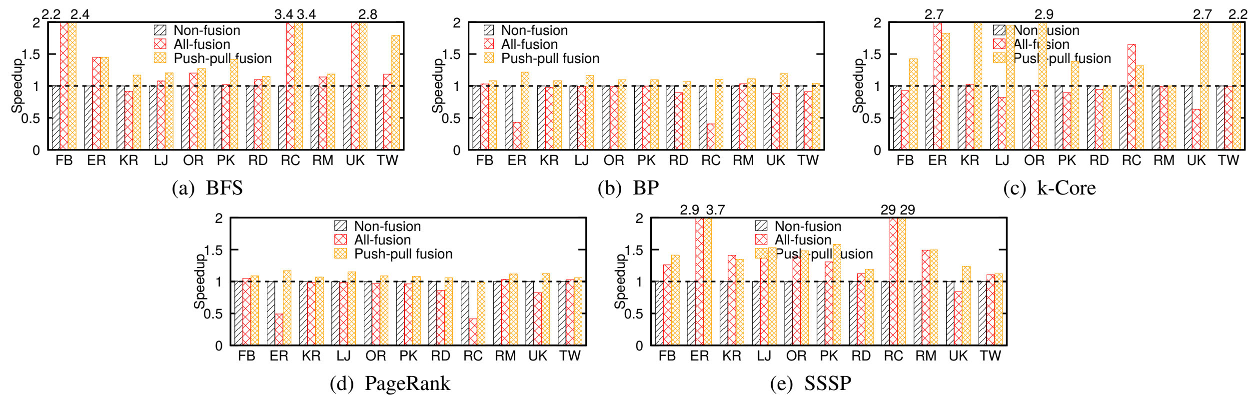


Figure 12: Benefit of push-pull based kernel fusion, normalized to the performance of no fusion.

erator optimizations [76, 41, 50, 33, 43, 59, 64, 13]. This section covers relevant work from three aspects: programming model, task management and kernel fusion.

Recently, we witness an array of graph analytical models. For instance, "think like a graph" [71] requires each vertex to obtain the view of the entire partition on one machine in order to minimize the communication cost. Furthermore, domain specific programming language systems, such as Galois [54], Green-Marl [23] and Trinity [63], allow programmers to write single-threaded source code while enjoying multi-threaded processing. In comparison, SIMD-X decouples the goal of programming simplicity and performance: with ACC, SIMD-X ultimately designs a data-parallel abstraction for deploying irregular graph applications on GPU. With JIT task management and push-pull based kernel fusion, SIMD-X is an order of magnitude faster than state-of-the-art CPU and GPU frameworks.

Task management is an important optimization for GPU-based graph computing. Besides batch filter [76, 50], there also exist other task management approaches – strided filter [41, 43] and atomic filter [46]. Particularly, strided filter resembles ballot filter but the former one experiences strided memory access when scanning the metadata thus performs up to 16× worse than ballot filter. Atomic filter relies is similar to online filter but it relies on atomic operation to put active vertices into global active list which suffers from orders of magnitude slow down than online filter. Besides ballot and online filter bests batch, stride and atomic filter, SIMD-X goes further via introducing a JIT controller to adaptively use online filter and ballot filter to further improve the performance. We also find that JIT task management can be exploited to help manage active lists for other applications such as warp segmentation [32] and CSR5 [44].

Kernel fusion affects applications far beyond graph computations. SIMD-X is closely related to global software barrier [79, 82]. However, previous work fails to identify the deadlock issue in this global software barrier problem, thus no solution towards this issue. In contrast, SIMD-X unveils, systematically analyzes, and resolves this problem. To avoid high register consumption, SIMD-X further selectively fuse kernels via exploiting the special kernel launching patterns of graph algorithms. It is also important to mention existing work [73] that only fuse kernels to barrier boundary. In comparison, SIMD-X fuses kernels across barriers. Our design can also benefit the popular Persistent Kernel [20] designs which have been found suffer from deadlock issues when the occupancy exceed an unknown bound [48, 24].

9 Conclusion

In this work, we propose SIMD-X, a parallel graph computing framework that supports programming and processing of *single instruction multiple, complex, data* on GPUs. Specifically, the Active-Compute-Combine (ACC) model provides ease of programming to programmers, while just-in-time task management and push-pull based kernel fusion leverage the opportunities for system-level optimization. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving significant speedup over the state-of-the-art.

Acknowledgment

The authors would like to thank the anonymous reviewers and Shepherd Chris Rossbach for their feedback and suggestions. Hang Liu did part of this work at the George Washington University. This work was partially supported by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774 at George Washington University and CRII Award No. 1850274 at University of Massachusetts Lowell. We also would like to gracefully acknowledge the support from XSEDE supercomputers and Amazon AWS, as well as the NVIDIA Corporation for the donation of the Titan Xp and Quadro P6000 GPUs to the University of Massachusetts Lowell.

References

[1] Nvidia cuda c programming guide. *NVIDIA Corporation*, 2011.

[2] 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 *OSDI*, volume 16, pages 265–283, 2016.

[3] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, 2017.

[4] S Beamer, K Asanovic, and D Patterson. Direction-optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10. IEEE, 2012.

[5] Bibek Bhattarai, Hang Liu, and H Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, volume 19, 2019.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, 2004.

[7] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 215–226. ACM, 2014.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] 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. ACM, 2012.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[11] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *28th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 349–359. IEEE, 2014.

[12] European Open Stream Map. `http://download.geofabrik.de/europe-latest.osm.bz2`,.

[13] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. Dr. BFS: Data Centric Breadth-First Search on FPGAs. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 208. ACM, 2019.

[14] Anil Gaihre, Yan Luo, and Hang Liu. Do Bitcoin Users Really Care About Anonymity? An Analysis of the Bitcoin Transaction Graph. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1198–1207. IEEE, 2018.

[15] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the international symposium on High-performance parallel and distributed computing (HPDC)*. ACM, 2019.

[16] Benedict R Gaster and Lee Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 2012.

[17] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Practical Recommendations on Crawling Online Social Networks. *IEEE Journal on Selected Areas in Communications*, 2011.

[18] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[19] GTgraph: A suite of synthetic random graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/`.

[20] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[21] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[22] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of international conference on Knowledge discovery and data mining (SIGKDD)*, pages 77–85, 2013.

[23] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 40, pages 349–362, 2012.

[24] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous-race-free memory models. *ACM SIGARCH Computer Architecture News*, 42(1):427–440, 2014.

[25] Yang Hu, Hang Liu, and H Howie Huang. High-Performance Triangle Counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2018.

[26] Yang Hu, Hang Liu, and H Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[27] H Howie Huang and Hang Liu. Big data machine learning and graph analytics: Current state and future challenges. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 16–17. IEEE, 2014.

[28] Yuede Ji, Hang Liu, and H Howie Huang. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 731–742. IEEE, 2018.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[30] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[31] Farzad Khorasani. *High Performance Vertex-Centric Graph Analytics on GPUs*. PhD Dissertation: University of California, Riverside, 2016.

[32] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. Scalable simd-efficient graph processing on gpus. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 39–50. IEEE, 2015.

[33] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[34] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 447–461. ACM, 2016.

[35] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 71. IEEE Press, 2016.

[36] Pradeep Kumar and H Howie Huang. Falcon: scaling IO performance in multi-SSD volumes. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, pages 41–53. USENIX Association, 2017.

[37] Pradeep Kumar and H Howie Huang. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, 2019.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 31–46. USENIX Association, 2012.

[40] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300. USENIX Association.

[41] Hang Liu and H. Howie Huang. Enterprise: Breadth-First Graph Traversal on GPU Servers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[42] Hang Liu and H. Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017.

[43] Hang Liu, H Howie Huang, and Yang Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[44] Weifeng Liu and Brian Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. 2010.

[46] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[47] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[48] Sepideh Maleki, Annie Yang, and Martin Burtscher. *Higher-order and tuple-based massively-parallel prefix sums*, volume 51. ACM, 2016.

[49] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[50] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *PPoPP*, 2012.

[51] Ulrich Meyer and Peter Sanders. Δ-Stepping: A Parallel Single Source Shortest Path Algorithm. *Algorithms—ESA'98*, 1998.

[52] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 2015.

[53] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[54] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471. ACM, 2013.

[55] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636. ACM, 2018.

[56] Nvidia. NVIDIA Kepler GK110 Architecture Whitepaper. 2013.

[57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[58] 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.

[59] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of USENIX conference on Annual Technical Conference*. USENIX Association, 2012.

[60] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[62] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: processing large-scale graphs on accelerator-based systems. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[63] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 505–516, 2013.

[64] Zachary Sherer, Eric Finnerty, Yan Luo, and Hang Liu. Software Hardware Co-Optimized BFS on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–190. ACM, 2019.

[65] 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) 16)*, pages 317–332.

[66] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[67] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[68] SNAP: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

[69] Tyler Sorensen, Alastair F Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable inter-workgroup barrier synchronisation for GPUs. In *ACM SIGPLAN Notices*, volume 51, pages 39–58. ACM, 2016.

[70] The University of Florida: Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[71] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From Think Like a Vertex to Think Like a Graph. *Proceedings of the VLDB Endowment*, 2013.

[72] Stanley Tzeng, Anjul Patney, and John D Owens. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010.

[73] Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[74] Kai Wang and Zhendong Su. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC.

[75] Siyuan Wang, Chang Lou Lou, Rong Chen, and Haibo Chen. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[76] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 265–266. ACM, 2015.

[77] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: GPU Graph Analytics. *arXiv preprint arXiv:1701.01170*, 2017.

[78] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. G ra M: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[79] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.

[80] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *ACM SIGPLAN Notices (PPoPP)*, volume 50, pages 194–204. ACM, 2015.

[81] Da Yan and Hang Liu. Parallel graph processing. *Encyclopedia of Big Data Technologies*, pages 1–8, 2018.

[82] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *PPoPP*, 2013.

[83] Jialing Zhang, Xiaoyan Zhuo, Aekyeung Moon, Hang Liu, and Seung Woo Son. Efficient Encoding and Reconstruction of HPC Datasets for Checkpoint/Restart. In *IEEE Symposium on Mass Storage Systems and Technologies*, 2019.

[84] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. *ACM SIGPLAN Notices (PPoPP)*, 50(8):183–193, 2015.

[85] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the Hidden Dimension in Graph Processing. In *OSDI*, pages 285–300, 2016.

[86] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 608–621. ACM, 2018.

[87] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[88] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 614–630. ACM, 2017.

[89] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data),*, pages 293–302. IEEE, 2017.

[90] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58. USENIX Association, 2015.

[91] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1543–1552, 2014.

[92] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386. USENIX Association, 2015.

LUMOS: Dependency-Driven Disk-based Graph Processing

Keval Vora
School of Computing Science
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Abstract

Out-of-core graph processing systems are well-optimized to maintain sequential locality on disk and minimize the amount of disk I/O per iteration. Even though the sparsity in real-world graphs provides opportunities for out-of-order execution, these systems often process graphs iteration-by-iteration, hence providing Bulk Synchronous Parallel (*synchronous* for short) mode of processing which is also a preferred choice for easier programmability. Since out-of-core setting limits the view of entire graph and constrains the processing order to maintain disk locality, exploiting out-of-order execution while simultaneously providing synchronous processing guarantees is challenging. In this paper we develop a generic dependency-driven out-of-core graph processing technique, called LUMOS, that performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees. Our cross-iteration value propagation technique identifies future dependencies that can be safely satisfied, and actively computes values across those dependencies without sacrificing disk locality. This eliminates the need to load the corresponding portions of graph in future iterations, hence reducing disk I/O and accelerating the overall processing.

1 Introduction

Disk-based processing of large graphs enables processing to scale beyond the available main memory in both single machine [1, 8, 9, 13, 14, 16, 17, 24, 34, 35, 38] and cluster based [23] processing environments. With limited amount of main memory available for processing, out-of-core graph systems first divide the graph into partitions that reside on disk, and then process these partitions one-by-one by *streaming* through them, i.e., by sequentially loading them in memory and immediately processing them. As expected, such out-of-core processing is I/O intensive and systems often spend significant amount of time in loading the partitions from disk; for example, GridGraph [38], a recent state-of-art out-of-core graph processing system, spends 69-90% of time in loading edges from disk partitions.

A common concern across graph processing systems is the nature of consistency semantics they offer for programmers to correctly express their graph algorithms. Consistency semantics in the context of iterative graph processing fundamentally decide when should a vertex's value (that is computed in a given iteration) become visible to its outgoing neighbors. The most popular consistency semantics is offered by the Bulk Synchronous Parallel (BSP) [27] model (hereafter called *synchronous* processing semantics) that separates computations across iterations such that vertex values computed in a given iteration become visible to their outgoing neighbors in the next iteration, i.e., values in a given iteration are computed based on values from the previous iteration. Such clear separation between values being generated v/s values being used allows programmers to clearly reason about the important convergence and correctness properties. Hence, synchronous processing semantics often becomes a preferred choice for large-scale graph processing [18, 24, 36, 38].

While out-of-core graph processing systems that provide synchronous processing semantics have been well-optimized to maintain sequential disk locality and to minimize the amount of disk I/O per iteration, they process graphs iteration-by-iteration such that processing for a given iteration starts only after all the partitions have been processed for the corresponding previous iteration. Such synchronous processing enforces dependencies between all values across subsequent iterations. However, dependencies in graph computation are determined by the structure of the input graph, and real-world graphs are often large and sparse. This means, more often than not, two randomly chosen vertices will not be directly connected to each other, hence deeming the dependency between their values to be unnecessary in synchronous processing. This sparsity in edges provides an opportunity to perform out-of-order execution such that unrelated values across multiple iterations get simultaneously computed to amortize the disk I/O cost across multiple iterations. However, achieving such out-of-order execution in an out-of-core setting without sacrificing sequential disk locality, as well as simultaneously providing synchronous processing guarantees is challenging.

In this paper, we develop a *dependency-aware cross-iteration value propagation technique* called **LUMOS** to enable future value computations that reduce disk I/O while still guaranteeing synchronous processing semantics. We refine vertex computations into two key components: the first step performs concurrent aggregation of incoming vertex values, and the next step uses the result of concurrent aggregation to compute the final vertex value. Upon doing so, we identify that computing aggregations requires all incoming vertex values to be available which is a strong precondition that limits future computations. However, values can be safely propagated to compute *partial aggregations* that lifts off the precondition of requiring all incoming vertex values. When the partial aggregations receive all required values, they can be used to compute the final future vertex values which can be propagated further down across subsequent future iterations.

We enable such cross-iteration value propagation across partitions as partition boundaries become natural points to capture the set of value dependencies that can be safely satisfied. We further increase cross-iteration propagation via locality-aware intra-partition propagation to exploit the inherent locality in real-world graphs which has been identified in recent works [36]. While LUMOS can also correctly process asynchronous algorithms (e.g., traversal algorithms like shortest paths), we further optimize LUMOS for asynchronous algorithms by exposing relaxed processing semantics in its processing model. Finally, to achieve maximum benefits we enhance LUMOS with several key out-of-core processing strategies like selective scheduling and light-weight partitioning that have been shown to be seminal in extracting performance in out-of-core processing.

While our dependency aware cross-iteration propagation model is general enough to be incorporated in any synchronous out-of-core graph processing system, we develop LUMOS by extending GridGraph which is a state-of-art out-of-core graph processing system that guarantees synchronous processing semantics. Our evaluation shows that LUMOS is able to compute future values across 71-97% of edges which eliminates the corresponding amount of disk I/O across those iterations, and hence, LUMOS is 1.8× faster than GridGraph while it still retains the same synchronous processing semantics. To the best of our knowledge, this is the first out-of-core graph processing technique that enables future value computation across iterations, while still retaining the synchronous processing semantics throughout all the iterations, which is crucial for easy programmability.

2 Background & Motivation

We first discuss about semantics of synchronous execution, and then summarize out-of-core graph processing techniques.

2.1 Synchronous Processing Semantics

The Bulk Synchronous Parallel (BSP) model [27] is a popular processing model that provides synchronous stepwise execution semantics with separated computation and communication/synchronization phases. Under the BSP model, values in a given iteration are computed based on values from the previous iteration. We illustrate the *synchronous processing semantics* of BSP model using the PageRank algorithm as an example [1] in Algorithm 1. The algorithm computes vertex values (`newPr`) using the ones computed in previous iteration (`pr`) as shown on line 6. The flow of values across iterations is explicitly controlled via SWAP() on line 11.

Algorithm 1 Synchronous PageRank

```
1: G = (V, E)                          ▷ Input graph
2: pr = {1, 1, ..., 1}                 ▷ Floating-point array of size |V|
3: while not converged do
4:     newPr = {0, 0, ..., 0}          ▷ Floating-point array of size |V|
5:     par-for (u, v) ∈ E do
6:         ATOMICADD(&newPr[v], pr[u] / |out_neighbors(u)|)
7:     end par-for
8:     par-for v ∈ V do
9:         newPr[v] = 0.15 + 0.85 × newPr[v]
10:    end par-for
11:    SWAP(pr, newPr)
12: end while
```

Such clear separation of values being generated v/s values being used allows programmers to clearly reason about the important convergence and correctness properties. Hence, synchronous processing semantics often becomes a preferred choice for large-scale graph processing [18, 24, 36, 38].

2.2 Out-of-Core Graph Processing

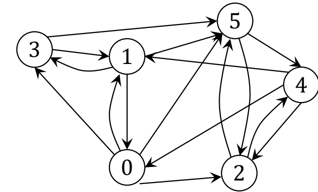
Disk-based graph processing has been a challenging task due to ever growing graph sizes. The key components in efficient out-of-core graph processing systems is a disk-friendly partition-based data-structure, and an execution engine that processes the graph in a partition-by-partition fashion that maximizes sequential locality. Figure 1 shows how a given graph is represented as partitions on disk. Each partition represents incoming edges for a range of vertices (*chunk*-based partitioning [38]); partition p_0 holds incoming edges for vertices 0 and 1, p_1 holds for vertices 2 and 3, and p_3 for vertices 4 and 5. The iterative engine processes the graph by going through these partitions in a fixed order; it sequentially loads edges from partition p_0 to process them in memory, then from partition p_1, and finally from the last partition p_2. Once all the partitions are processed, the iteration ends by performing computations across vertex values, that may reside on disk or in-memory (depending on availability of memory). This entire process is repeated for multiple iterations until algorithm-specific termination condition is satisfied.

Several works aim to improve out-of-core graph processing [1, 8, 9, 13, 14, 16, 17, 24, 31, 32, 34, 35, 38] as summarized in Table 1. Depending on the processing semantics they offer, they fall in two categories:

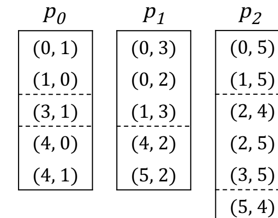
[1] Algorithm 1 is simplified to eliminate details like selective scheduling.

	Out-of-Order Execution	Future Value Computation	Synchronous Semantics	Async. Algorithms (e.g., SSSP)
GraphChi [13]	✗	✗	✓	✓
X-Stream [24]	✗	✗	✓	✓
GridGraph [38]	✗	✗	✓	✓
FlashGraph [35]	✗	✗	✓	✓
TurboGraph [8]	✗	✗	✓	✓
Mosaic [17]	✗	✗	✓	✓
GraFBoost [9]	✗	✗	✓	✓
Graphene [15]	✗	✗	✓	✓
Garaph [16]	✗	✗	✓	✓
DynamicShards [31]	✓	✗	✓	✓
Wonderland [34]	✓	✓	✗	✓
CLIP [1]	✓	✓	✗	✓
AsyncStripe [4]	✓	✓	✗	✓
LUMOS	✓	✓	✓	✓

Table 1: Key characteristics of existing out-of-core graph processing systems and **LUMOS**.



(a) Example graph.



(b) Edge partitions.

Figure 1: An example graph partitioned into three partitions (p_0, p_1 and p_2) that reside on disk.

(A) Synchronous Out-of-Core Graph Processing. GraphChi [13], X-Stream [24], GridGraph [38], and others provide synchronous processing semantics. While initial systems like GraphChi [13] and X-Stream [24] proposed efficient processing models, their performance is limited due to management of edge-scale intermediate values in every iteration (edge values in GraphChi and edge updates in X-Stream).

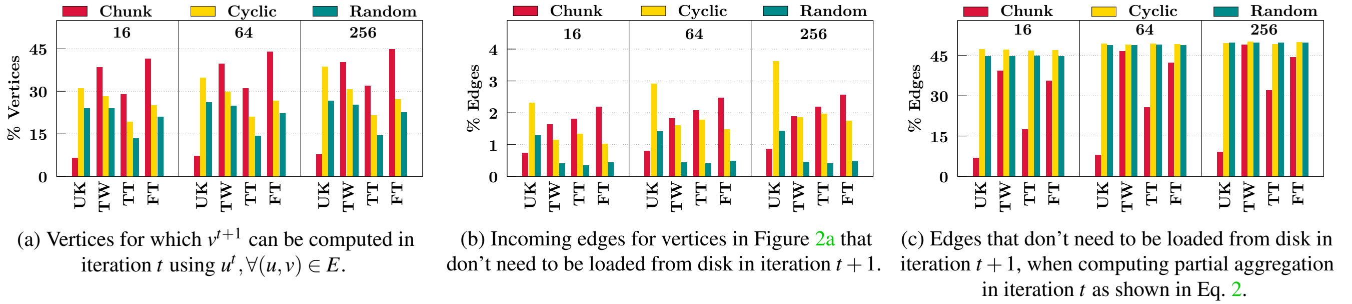
GridGraph [38] is an efficient out-of-core graph processing system that eliminates edge-scale intermediate updates. It divides vertices into subsets called *chunks* and partitions edges into 2D grid based on these chunks. In Figure 1b, the dashed-horizontal lines represent boundaries of *blocks* such that the entire representation becomes a 2D grid. The 2D grid is processed by streaming through edge-blocks. Furthermore, GridGraph enables selective scheduling which eliminates unnecessary edges to be loaded from disk by skipping partitions. Since its processing model is designed to minimize disk I/O, it is the state-of-art out-of-core graph processing system that provides synchronous processing semantics.

(B) Asynchronous Out-of-Core Graph Processing. Recent works like CLIP [1] and Wonderland [34] are customized for asynchronous algorithms like path-based algorithms (e.g., BFS and shortest paths). These frameworks leverage the algorithmic properties (e.g., monotonicity [28, 29]) to process partitions based on newly computed values, resulting in faster convergence. CLIP [1] processes partitions multiple times in memory while Wonderland [34] performs abstraction-guided processing for faster information propagation. Even though these techniques perform out-of-order computations (see Table 1), they do not provide synchronous processing semantics since they violate the processing order across computed vertex values. Hence, they cannot be used for synchronous graph algorithms.

Limitations with Out-of-Core Systems. As shown in Table 1, none of the systems perform *future value computation* (i.e., beyond a single iteration) while simultaneously providing synchronous processing semantics. In synchronous out-of-core frameworks, the processing model is tied down to strict iteration-by-iteration processing. Such tight coupling between synchronous semantics and strict processing order limits the performance of out-of-core graph processing. Particularly, the sparsity in real-world graphs often presents opportunities to proactively compute future values based on when value dependencies get resolved; realizing such processing across future iterations can be beneficial in out-of-core setting since edges corresponding to values that have already been computed for future iterations do not need to be loaded in those iterations, hence directly reducing disk I/O. However, such acceleration via future value computation is not achieved in out-of-core systems due to their strict iteration-by-iteration processing.

It is crucial to note that out-of-order execution does not necessarily result in computing across future values. Specifically, DynamicShards [31] performs out-of-order execution to dynamically capture the set of active edges to be processed in a given iteration. It achieves this by dropping inactive (or useless) edges across iterations and delaying computations that cannot be performed due to missing edges. While the delayed computations get periodically processed in shadow iterations (i.e., out-of-order execution), they do not compute across future values to leverage sparse dependencies. Asynchronous systems [1,34], on the other hand, do compute beyond a single iteration, but do not provide processing semantics.

This poses an important question: *how to process beyond a single iteration to reduce disk I/O in out-of-core processing while simultaneously guaranteeing synchronous processing semantics?*



(a) Vertices for which v^{t+1} can be computed in iteration t using $u^t, \forall(u,v) \in E$.

(b) Incoming edges for vertices in Figure 2a that don't need to be loaded from disk in iteration $t+1$.

(c) Edges that don't need to be loaded from disk in iteration $t+1$, when computing partial aggregation in iteration t as shown in Eq. 2.

Figure 2: Percentage of vertex computations and edge savings across three light-weight partitioning strategies (chunking, cyclic hashing and random partitioning) and three partition sizes (#partitions = 16, 64 and 256) on four large graphs (UK, TW, TT and FT from Table 4).

3 LUMOS: Dependency-Driven Cross-Iteration Value Propagation

Since our goal is to provide synchronous processing semantics while overlapping computations across multiple iterations, we first characterize cross-iteration dependencies to capture synchronous semantics and then develop our out-of-core value propagation strategy that guarantees those captured semantics.

3.1 Characterizing Synchronous Dependencies

Synchronous iterative graph algorithms compute vertex values in a given iteration based on values of their incoming neighbors that were computed in the previous iteration. Since computations are primarily based on graph structure, such cross iteration dependencies can be captured via the graph structure as follows:

$$\forall(u,v) \in E, \;\; u^t \mapsto v^{t+1} \tag{1}$$

where u^t and v^{t+1} represent values of vertex u in iteration t and vertex v in iteration $t+1$ respectively, and $\mapsto$ indicates that v^{t+1} is value-dependent on u^t. It is important to note that there are no dependency relationships among vertices that are not directly connected by an edge. With limited view of graph structure available at any given time in out-of-core graph processing, the cross-iteration dependencies get satisfied by processing vertices and edges in a given iteration only after processing for the previous iteration is completed.

3.2 Out-of-core Value Propagation

We ask two important questions that allow us to identify the interplay between cross-iteration value propagations to satisfy future dependencies and partition-by-partition processing orchestrated by out-of-core graph systems.

3.2.1 When to propagate?

A straightforward way to enable processing beyond a given single iteration t is to compute vertex values for the subsequent iteration $t+1$ if incoming neighbors' values corresponding to t are available at the time when those vertices are processed. With partition-by-partition out-of-core processing, we know that values for vertices belonging to a given partition p become available when p is processed; hence, we can allow outgoing neighbors to use these available values and compute for subsequent iterations if their partitions get processed after p in the same iteration. While theoretically this appears to be a promising direction, we profile the large graphs from Table 4 to measure the number of vertices for which future values can be computed (shown in Figure 2a), and the number of edges that don't need to be loaded for the corresponding vertices in future iterations (shown in Figure 2b). To eliminate the impact of partitioning, we profiled across three light-weight partitioning schemes, chunk-based partitioning (as used in [38]), cyclic partitioning (where vertex ids are hashed to partitions) and random partitioning; and, across three partition sizes corresponding to number of partitions being 16, 64 and 256. Even though Figure 2a shows up to 45% of vertices can compute values for subsequent iterations, it contributes to only 1-4% of edge savings as shown in Figure 2b. This means, future values can be computed for vertices that have low in-degree and, as expected, high in-degree vertices cannot compute future values since values for all of their incoming neighbors do not become available in time.

To achieve high amount of cross-iteration value propagation, we want to relax the precondition such that availability of *all* incoming neighbor values does not become a requirement. We achieve this by computing only the aggregated values for future iterations instead of computing final vertex values. Let $\bigoplus$ denote the aggregation operator that computes the intermediate value based on incoming neighbors and f denote the function to compute vertex's value based on aggregated intermediate value. For example, in PageRank (Algorithm 1), ATOMICADD on line 6 represents $\bigoplus$ and line 9 shows f. In a given iteration t, we aim to compute [2]:

$$v^t = f\Big(\bigoplus_{\forall e=(u,v)\in E} (u^{t-1})\Big) \quad \text{and} \quad g(v^{t+1}) = \bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\;\; p(u)<p(v)}} (u^t) \tag{2}$$

[2] Values residing on edges (i.e., edge weights) have been left out from equations for simplicity as they do not impact cross-iteration dependencies.

where $g(v)$ represents aggregated value of v and $p(v)$ is the partition to which v belongs. It is important to note that $\bigoplus_{\forall e=(u,v)\in E}$ represents a complete aggregation while $\bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u)<p(v)}}$ represents partial aggregation as the precondition $p(u) < p(v)$ may not be satisfied by all edges. Since $\forall u \in V, u^{t-1}$ is available in iteration t (due to barrier semantics), we can perform complete aggregation and also compute the vertex's value using f. However, since u^t becomes available as partitions get processed in the same iteration t, at a given point in time only the available u^t values can be propagated to compute the partial aggregation which satisfies the future cross-iteration dependency $u^t \mapsto v^{t+1}$. Since typically partitions get processed in numerical order, with P being total number of partitions, we know that $\forall i, j \in [0,P)$, if $i < j$, partition i gets processed before partition j. This ordering is captured in the precondition for partial aggregation $g(v^{t+1})$. Hence, in Figure 1, as p_0, p_1 and p_2 get processed in that order, v_0^t is available for $g(v_2^{t+1})$, $g(v_3^{t+1})$ and $g(v_5^{t+1})$ via $(0,2)$, $(0,3)$ and $(0,5)$ respectively, while v_3^t is not available for $g(v_1^{t+1})$ during iteration t.

As shown in Figure 2c, the percentage of values propagated (via partial aggregation) increases to 40-50%; the edges corresponding to these propagations need not be loaded in the subsequent iteration (i.e., directly reducing disk I/O), which is significantly higher compared to that in Figure 2b. We also observe that random partitioning compares well with other techniques and enables higher cross-iteration propagation; this can be reasoned with the high chances of an edge (u,v) being placed across partitions such that $p(u) < p(v)$. In Section 5.3, we will explore more light-weight partitioning strategies that will further enable cross-iteration propagation.

Note that the above value propagation requires vertex values for the current iteration to be computed as partitions get processed. We developed our processing model to simultaneously compute vertex values as corresponding partition's edges get processed (discussed further in Section 5.1).

3.2.2 How far (in iteration space) to propagate?

Cross-iteration dependencies are linear in iteration space and hence, we can potentially propagate values for future iterations beyond $t+1$. In order to guarantee synchronous processing semantics, we need to ensure that v^{t+1} gets computed in iteration t before it is further propagated to out-neighbors of v. We define a value v^x to be *computable* when all its incoming values corresponding to iteration $x-1$ have been propagated to $g(v^x)$ (i.e., a complete aggregation has occurred in $g(v^x)$). Since out-of-core processing propagates values to vertices based on its partitions, values can become computable when the corresponding vertex's partition gets processed. Computable values get computed by applying f on $g(v^x)$ to achieve v^x which can be further propagated to out-neighbors of v for $x+1$. For example in Figure 1, vertex 3 has two incoming edges $(0,3)$ and $(1,3)$, both of which contribute to $g(v_3^{t+1})$ during iteration t; hence, we can compute $v_3^{t+1} = f(g(v_3^{t+1}))$ during iteration t itself and further propagate v_3^{t+1} across the outgoing edge $(3,5)$ in the same iteration t.

	UK	TW	TT	FT
$D=2$	6.8 - 49.7	39.2 - 50.1	17.4 - 49.7	35.6 - 49.8
$D=3$	0.38 - 0.52	0.13 - 0.25	0.11 - 0.44	0.04 - 0.26
$D=4$	< 0.01	< 0.01	< 0.01	< 0.01

Table 2: Percentage (min-max range) of edge propagations across three partitioning strategies from Figure 2.

For a given iteration t, we know that v^t becomes computable when $p(v)$ gets processed. Hence, for any arbitrary k, v^{t+k} becomes computable when $\forall(u,v) \in E, p(u) < p(v)$ and u^{t+k-1} is computable. This is because the partial aggregation $\bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u)<p(v)}}$ becomes equivalent to complete aggregation $\bigoplus_{\forall e=(u,v)\in E}$ when $\forall(u,v) \in E, p(u) < p(v)$. These computable values that get further propagated lead to I/O reduction in the corresponding future iterations. We define distance of propagation (in iteration space) based on the difference between the current iteration number and the iteration number for which any value propagation occurs in the current iteration. Formally, the *distance of propagation D* is defined as:

$$D = \max_{\forall t} \left(\max_{\forall g(v^{t+k})} (k+1) \right)$$

In traditional out-of-core graph systems, barriers across iterations ensure that $D = 1$. Our dependency-aware cross-iteration propagation achieves $D > 1$; for example, with propagation for immediately subsequent iteration, D is 2. Table 2 summarizes the percentage of edge propagations that occur across distances 2, 3 and 4. As we can see, propagations decrease drastically at distance 3, and are close to 0% after that. Since achievable benefits are minor beyond propagation distance 2, we perform cross-iteration propagation until distance 2, i.e., for the current iteration t and the next iteration $t+1$.

3.3 Graph Layout

Since cross iteration dependencies are primarily based on the input graph structure, value propagations beyond current iteration can be statically determined based on the precondition involved in Eq. 2. In order to completely avoid reading edges whose dependencies have been satisfied, we create separate graph layouts for subsequent iterations. In this way, the execution switches between different graph layouts on disk. In theory, we can create D separate graph layouts to propagate values across D iterations; however, to simplify exposition we discuss the layout for $D = 2$ since larger propagation distances provide diminishing benefits (as discussed in Section 3.2). With $D = 2$, we have two graph layouts: the *primary* layout consisting of all edges, and the *secondary* layout containing only subset of edges. Figure 3 shows the secondary layout and its corresponding graph representation for the primary graph

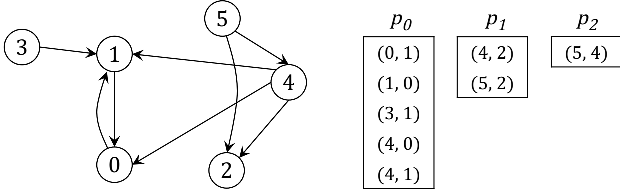


Figure 3: Secondary layout for graph from Figure 1.

Graph	Without Intra-Partition Propagation	With Intra-Partition Propagation
UK	6.84 - 49.69	49.76 - 92.47
TW	39.24 - 50.09	50.31 - 62.22
TT	17.43 - 49.74	43.32 - 61.87
FT	35.58 - 49.80	47.20 - 56.19

Table 3: Percentage (min-max range) of edge propagations with and without intra-partition propagation across three partitioning strategies from Figure 2.

layout in Figure 1. The secondary layout does not contain 8 out of 16 edges since the corresponding dependencies get satisfied while processing primary layout.

We analyze the reduction in I/O caused by our secondary graph layout. Since $|E| >> |V|$, out-of-core graph processing systems like GridGraph achieve high performance by ensuring that the edges are loaded only once in every iteration. Hence, for each iteration, their I/O amount is $|E| + k \times |V|$ where $k \geq 2$ captures batch loading of vertices for each partition. While our I/O amount for the iteration using primary layout remains the same, it reduces significantly when secondary layout is used. Let α be the ratio of edges for which values are propagated for subsequent iteration when primary layout is processed. Since those edges are not present in secondary layout, the I/O amount directly reduces by $\alpha \times |E|$. Hence, the total I/O amount is:

$$C = \begin{cases} (1-\alpha) \times |E| \; + \; k \times |V| & \ldots \text{ secondary layout} \\ |E| \; + \; k \times |V| & \ldots \text{ primary layout} \end{cases}$$

As we will see in Section 6, α is typically over 0.7 which drastically reduces the I/O amount.

3.4 Intra-Partition Propagation

So far cross-iteration propagation is performed for edges across partitions as dictated by Eq. 2. While this captures a large subset of edges that don't need to be loaded in the subsequent iteration, it has been recently shown in [36] that real-world graphs often possess natural locality such that adjacent vertices are likely to be numbered close to each other due to the traversal nature of graph collection strategies (e.g., crawling, anonymization, etc.). This means, chunk-based partitioning strategies where contiguous vertex-id ranges get assigned to partitions have several edges such that both end-points belong to the same partition.

In order to leverage this natural locality, in a given iteration t, we aim to propagate value for subsequent iteration across edge (u, v) where $p(u) = p(v)$. However, since u^t becomes available as partition $p(u)$ gets processed in the same iteration t, u^t can only be propagated after $p(u)$ has been processed. Hence, if (u, v) can be held in memory until $p(u)$ gets fully processed, we can propagate u^t to satisfy the future cross-iteration dependency $u^t \mapsto v^{t+1}$. This means, intra-partition cross-iteration propagation relaxes our precondition for partial aggregation to become:

$$g(v^{t+1}) = \bigoplus_{\substack{\forall e=(u,v)\in E \\ s.t.\ p(u) \leq p(v)}} (u^t) \tag{3}$$

Table 3 summarizes the increase in cross-iteration propagation when intra-partition propagation is enabled. As we can see, cross-iteration propagation increases to 43-92% with intra-partition propagation, which further saves disk I/O.

We enable intra-partition cross-iteration value propagation by ensuring that partition sizes remain small enough such that they can be entirely held in memory. As shown in Section 5.1, our processing model holds the partition in memory until it gets processed, and then performs cross-iteration propagation for edges whose both end-points belong in the partition.

3.5 Value Propagation v/s Partition Size

The amount of cross-iteration value propagation not only depends on the partitioning strategy, but also varies based on the size of partitions. As we can observe in Figure 2c, the same partitioning strategy with different partition sizes enables different amount of cross-partition propagation. In general, since cross-partition propagation is achieved on edges crossing partition boundaries, smaller partitions lead to higher cross-partition propagation compared to that by larger partitions. As an extreme case, each vertex residing in its own partition would result in smallest partitions, and would cause highest amount of cross-partition value propagation.

Intra-partition propagations, on the other hand, are amenable to large partitions as they enable propagations across edges within the partitions. Even though smaller partitions allow lesser intra-partition propagations compared to that allowed by larger partitions, the set of edges crossing smaller partitions is larger, hence allowing cross-partition propagations through edges that get missed by intra-partition propagations. It is interesting to note that as partitions become smaller, edges that were originally participating in intra-partition propagation get cut across partition boundaries either in forward direction (i.e., $p(u) < p(v)$ for edge (u, v)) or in backward direction. Since cross-partition value propagation occurs on forward edges only, the overall cross-iteration value propagation (combined cross-partition and intra-partition) reduces as partitions become smaller. Hence, it is preferable to have large partitions that can fit in main memory to maximize cross-iteration value propagation. In Section 5.3, we propose light-weight partitioning heuristics that increase the amount of cross-iteration value propagation, which in turn reduces the severity of partition size's impact.

3.6 LUMOS with DynamicShards

LUMOS can be combined with DynamicShards [31] to perform cross-iteration value propagation with dynamic edge selection. Specifically, primary layout partitions can be made dynamic (by dropping inactive edges) while secondary layout partitions can be left static since they are already small in size. Furthermore, shadow iterations require the missing graph edges to process delayed computations, and hence, shadow iterations must be scheduled during primary iterations in order to amortize loading costs. It is interesting to note that even though secondary layouts are kept static, computations from secondary layouts can get delayed due to transitive delays occurring from previous iterations' primary layouts.

4 LUMOS for Asynchronous Algorithms

Several graph analytics algorithms like traversal based monotonic algorithms (e.g., shortest paths, BFS, connected components, etc.) are asynchronous, i.e., they do not require synchronous processing semantics to guarantee correctness [28, 30]. Nevertheless, having synchronous semantics does not hurt the correctness guarantees for asynchronous algorithms because synchronous execution is one of the legal executions under the set of asynchronous executions. In other words, the dependencies enforced by synchronous processing semantics in Eq. 1 are only stricter (while still being legal) than that required by asynchronous semantics. Since LUMOS's out-of-order value propagation technique does not violate the dependencies defined in Eq. 1, LUMOS works correctly on asynchronous algorithms like shortest paths and connected components as well.

Furthermore, LUMOS can be optimized to efficiently process asynchronous algorithms by incorporating relaxed processing semantics in its processing model. For example, traversal based monotonic algorithms often rely on selection functions [29] like MIN() that enable vertex's value to be computed based on value coming from its single incoming edge. This means, intermediate vertex values computed based on subset of their incoming values represent a valid final value (instead of a partial aggregation) that can be instantly propagated to outgoing edges. In other words, computations in Eq. 2 do not require separation of values across iterations t and $t+1$, and $g(v^t)$ can be directly incorporated in v^t. Hence, LUMOS can enable value propagation for asynchronous algorithms by maintaining a single version of vertex values and directly propagating the updated values across edges.

Such *asynchronous value propagation* is achieved both across (inter-partition) and within (intra-partition) partitions; intra-partition propagations are achieved by recomputing over in-memory partition as described in Section 3.4. An interesting side effect of asynchronous value propagation is that secondary graph layout is no longer necessary because any value propagation across edges in the primary graph layout does not violate asynchronous semantics, and hence, entire processing can occur on the primary graph layout itself.

5 The LUMOS System

So far we discussed value propagation in a generalized out-of-core processing context without focusing on any particular system. This makes our proposed cross-iteration value propagation techniques useful for existing out-of-core processing systems. We now discuss the important design details involved in incorporating LUMOS into GridGraph. We choose GridGraph since its streaming-apply processing model is designed to minimize I/O amount, making it state-of-art synchronous out-of-core graph processing system.

5.1 Propagation based Processing Model

LUMOS's processing model is similar to out-of-core processing systems where edges are loaded from disk in batches and processed in memory. We discuss the processing model at partition level to showcase how cross-iteration propagations are performed. Algorithm 2 shows how primary and secondary partitions are processed; LUMOS offers three key programming interfaces: (a) PROCESSPRIMARY performs standard propagation and cross-iteration propagation across primary partitions; (b) PROCESSSECONDARY performs standard propagations across secondary partitions; and, (c) VERTEXMAP performs updates across vertices. PROCESSPRIMARY processes both, edges (lines 4, 6, 13) and vertices (line 9). PROCESSSECONDARY's structure is kept similar to PROCESSPRIMARY to enable easier programmability of graph algorithms. The primary partitions get processed in even iterations using PROCESSPRIMARY while the secondary partitions get processed in odd iterations using PROCESSSECONDARY. Figure 4 illustrates how partition-by-partition processing is achieved using primary and secondary layouts across two consecutive iterations. Along with the traditional cross-iteration barriers, we also have cross-partition barriers while processing primary partitions to ensure that precondition in Eq. 2 gets correctly satisfied for cross-iteration propagation; these cross-partition barriers are not required while processing secondary partitions. Algorithm 3 shows PageRank algorithm using our propagation interface. Beyond the standard edge function PROPAGATE and vertex function COMPUTE, we also use the cross-iteration edge propagation function CROSSPROPAGATE. While the shape of CROSSPROPAGATE is similar to PROPAGATE, they operate on different values for the same edge, i.e., PROPAGATE aggregates for current iteration while CROSSPROPAGATE aggregates for subsequent iteration. The aggregation for subsequent iteration is made available using ADVANCE (line 13).

5.2 Selective Scheduling

One of the strengths of 2D grid layout is that it enables selective scheduling of edge computations so that edge-blocks can be skipped to reduce unnecessary I/O [38]. LUMOS carefully incorporates selective scheduling with cross-iteration propagation to ensure that scheduling gets correctly managed for primary and secondary partitions.

Algorithm 2 Propagation Interface

```
1:  function PROCESSPRIMARY(PROPAGATE,
                        CROSSPROPAGATE, COMPUTE)
2:    for partition ∈ primaryPartitions do
3:      par-for edge ∈ partition do
4:        PROPAGATE(edge)
5:        if p(edge.source) < p(edge.target) then
6:          CROSSPROPAGATE(edge)
7:        end if
8:      end par-for
9:      VERTEXMAP(COMPUTE, vertex_chunk(partition))
10:     if Locality-Aware Intra-Partition Propagation then
          /* partition is held in memory (see Section 3.4) */
11:       par-for edge ∈ partition do
12:         if p(edge.source) = p(edge.target) then
13:           CROSSPROPAGATE(edge)
14:         end if
15:       end par-for
16:     end if
17:   end for
18: end function
19: function PROCESSSECONDARY(PROPAGATE, COMPUTE)
20:   for partition ∈ secondaryPartitions do
21:     par-for edge ∈ partition do
22:       PROPAGATE(edge)
23:     end par-for
24:     VERTEXMAP(COMPUTE, vertex_chunk(partition))
25:   end for
26: end function
27: function VERTEXMAP(VFUNC, VS = V)          ▷ V is default arg.
28:   sum = 0
29:   par-for v ∈ VS do
30:     sum += VFUNC(v)
31:   end par-for
32:   return sum
33: end function
```

Algorithm 3 PageRank Example

```
1:  function PROPAGATE(e)
2:    ATOMICADD(&sum[e.target], pagerank[e.source] / outdegree[e.source])
3:  end function
4:  function CROSSPROPAGATE(e)
5:    ATOMICADD(&secondary_sum[e.target],
                sum[e.source] / outdegree[e.source])
6:  end function
7:  function COMPUTE(v)
8:    sum[v] = 0.15 + 0.85 × sum[v]
9:  end function
10: function ADVANCE(v)
11:   diff = |pagerank[v] − sum[v]|
12:   pagerank[v] = sum[v]
13:   sum[v] = secondary_sum[v]
14:   secondary_sum[v] = 0
15:   return diff
16: end function
17: pagerank = [1, ..., 1]
18: sum = [0, ..., 0]
19: secondary_sum = [0, ..., 0]
20: iteration = 0
21: converged = false
22: while ¬converged do
23:   if iteration % 2 == 0 then
24:     PROCESSPRIMARY(PROPAGATE, CROSSPROPAGATE,
                      COMPUTE);
25:   else
26:     PROCESSSECONDARY(PROPAGATE, COMPUTE);
27:   end if
28:   d = VERTEXMAP(ADVANCE);
29:   converged = d / |V| ≤ threshold
30:   iteration = iteration + 1
31: end while
```

An *active* edge-block represents edges that will be loaded from disk; otherwise, they will be skipped. When processing primary partitions (i.e., during even iterations), depending on the state (active/inactive) of an edge-block for primary and secondary layouts at the time when it needs to be processed, there can be four cases. While three of those cases can be handled in the same way as done for primary partitions, the case when the edge-block is inactive for primary layout but is active for secondary layout need to be considered carefully. In this case, while processing secondary partitions in the subsequent iteration, the corresponding edge-block gets loaded from the primary layout instead of secondary layout to ensure that all necessary edges within the edge-block participate correctly in value propagation. LUMOS maintains this cross-iteration selective scheduling information using 2-bits per edge-block; first bit indicating whether the edge-block is active or inactive, and second bit indicating whether to load edge-block from primary layout or secondary layout.

5.3 Graph Layout & Partitioning

For a graph $G = (V, E)$, V is divided into P disjoint subsets of vertices (called *chunks*), $C = \{c_0, c_1, ..., c_{P-1}\}$ such that $\bigcup_{c \in C} c = V$ and $\forall c_i, c_j \in C, c_i \cap c_j = \varnothing$. The edges are represented as a 2D grid of $P \times P$ edge-blocks on disk. An edge (u, v) is in edge-block b_{ij} if $u \in c_i \wedge v \in c_j$. It is important to note that a column i in the 2D grid has incoming edges for vertices belonging to c_i, and similarly a row j has outgoing edges for vertices belonging to c_j. Hence, each column is a partition for LUMOS in accordance with the precondition in Eq. 2 that satisfies the dependency relation. Similarly, each row is a partition for LUMOS when values need to be propagated across incoming edges (i.e., a transposed view).

As discussed in Section 3.3, we create a primary layout and a secondary layout. This means, we create two separate 2D grids on disk, one for each layout. An important issue in creating grid layouts is partitioning V into P chunks. Out-of-core processing systems use a simplified structure-oblivious partitioning strategy based on vertex-id ranges, i.e., assuming vertices are numbered between 0 to $|V|$, chunks are formed as contiguous range of vertex numbers: vertices 0 to $k-1$ form chunk 0, vertices k to $2k-1$ form chunk 1, and so on. While such structure-oblivious partitioning enables good amount of cross-iteration propagation (shown in Section 3), we develop greedy partitioning strategies that carefully use the vertex degree information to maximize cross-iteration value propagation.

Let $\mathscr{P} = \{p(v_0), p(v_1), ..., p(v_{|V|-1})\}$ capture the partition-

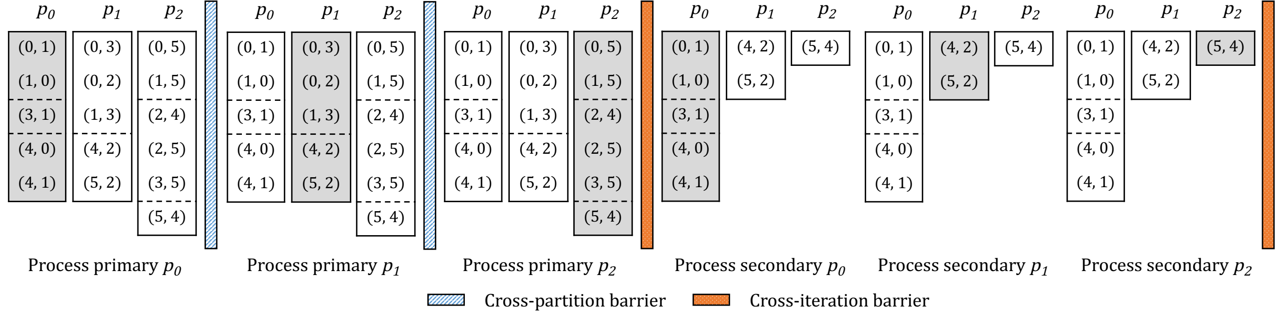


Figure 4: Execution using primary and secondary layouts across two consecutive iterations.

ing information that maps vertices in V to chunks in C. We define our partitioning objective as:

$$\underset{\mathscr{P}}{\arg\max}\ |\{(u,v) : (u,v) \in E \ \wedge\ p(u), p(v) \in \mathscr{P} \ \wedge\ p(u) < p(v)\}| \quad (4)$$

Note that with intra-partition propagation, the above condition becomes $p(u) \leq p(v)$ and there is an additional constraint to limit partition sizes:

$$\forall c_i \in C, \sum_{\substack{\forall v \in c_i \ \wedge \\ \forall (u,v) \in E}} (u,v) \ < \ T$$

where T is a threshold based on available memory.

Since the condition in Eq. 4 can be directly viewed as u's outgoing edges contributing to cross-iteration dependencies, a straightforward greedy heuristic can be to place vertices with higher out-degree in earlier partitions. However, an interesting dual to this reasoning can be that v's incoming edges contribute to cross-iteration dependencies and hence, the greedy heuristic can be to place vertices with higher in-degree in later partitions. Based on these insights, we develop three key partitioning heuristics to assign vertices in V to chunks in C:

(A) Highest Out-Degree First: $\forall u, v \in V$,
if $p(u) < p(v)$, then $out_degree(u) \geq out_degree(v)$

(B) Highest In-Degree Last: $\forall u, v \in V$,
if $p(u) < p(v)$, then $in_degree(v) \geq in_degree(u)$

(C) Highest Out-Deg. to In-Deg. Ratio First: $\forall u, v \in V$,
if $p(u) < p(v)$, then $\frac{out_degree(u)}{in_degree(u)} \geq \frac{out_degree(v)}{in_degree(v)}$

Note that the above heuristics are simpler than structure-based partitioning since computing degrees of vertices only requires a single pass over the edge list.

6 Evaluation

We evaluate LUMOS using billion-scale graphs and synchronous graph algorithms, and compare its performance with GridGraph [38] which is the state-of-art out-of-core graph processing framework that provides synchronous semantics.

Graphs	Disk Size	|E|	|V|
UKDomain (UK) [2]	6.9-20.7GB	1.0B	39.5M
Twitter (TW) [12]	11-33GB	1.5B	41.7M
TwitterMPI (TT) [3]	15-45GB	2.0B	52.6M
Friendster (FT) [5]	20-60GB	2.5B	68.3M
Yahoo (YH) [33]	50-150GB	6.6B	1.4B
RMAT29 (RM)	66-198GB	8.6B	537M

Table 4: Real world & synthetic graphs.

6.1 Experimental Setup

LUMOS's evaluation is carried out across three different AWS EC2 storage optimized instances. For performance experiments (Section 6.2), we use h1.2xlarge with 8 vCPUs, 32GB memory and 2TB HDD. Its disk sequential read bandwidth [3] is 278MB/sec whereas the memory subsystem read bandwidth is 9.6GB/sec. To study the effect of I/O scaling (Section 6.3), we use d2.4xlarge and i3.8xlarge instances. The d2.4xlarge instance is used with 16 vCPUs, 32GB memory and 1 to 4 2TB HDDs providing disk bandwidth of 195MB/sec to 768MB/sec, whereas the i3.8xlarge instance is used with 32 vCPUs, 64GB memory and 1 to 4 1.9TB SSDs providing disk bandwidth of 1.2GB/sec to 3.9GB/sec.

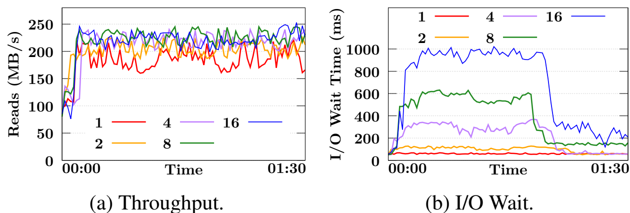
We use six synchronous graph algorithms. PageRank (PR) [22] is an algorithm to rank web-pages while Weighted Pagerank (WPR) is its variant where edges have weights as applied for social network analysis. Co-Training Expectation Maximization (CoEM) [21] is a semi-supervised learning algorithm for named entity recognition. Belief Propagation (BP) [10] is an inference algorithm to determine states of vertices based on sum of products. Label Propagation (LP) [37] is a learning algorithm while Dispersion (DP) [11] is a simulation based information dispersion model. We run each algorithm for 10 iterations; PR and DP operate on unweighted graphs while CoEM, LP, WPR and BP require weighted graphs. This adds 4 bytes per edge for CoEM, LP and WPR, and 16 bytes per edge for BP; hence, increasing graph sizes to 1.5× and 3× respectively.

We evaluate LUMOS using billion scale graphs from Ta-

[3] Disk sequential read bandwidth measured using `hdparm`.

	Version	TT	FT	YH
PR	GridGraph	737	1008	3223
	LUMOS-BASE	563	659	2027
	LUMOS	439	583	1885
	× LUMOS	**1.68×**	**1.73×**	**1.71×**
CoEM	GridGraph	1119	1554	5082
	LUMOS-BASE	861	1029	3216
	LUMOS	651	914	3043
	× LUMOS	**1.72×**	**1.70×**	**1.67×**
DP	GridGraph	846	1032	3484
	LUMOS-BASE	656	675	2219
	LUMOS	498	611	2111
	× LUMOS	**1.70×**	**1.69×**	**1.65×**
BP	GridGraph	2498	3782	13769
	LUMOS-BASE	1921	2456	8660
	LUMOS	1487	2212	7913
	× LUMOS	**1.68×**	**1.71×**	**1.74×**
WPR	GridGraph	984	1302	4330
	LUMOS-BASE	769	874	2758
	LUMOS	569	770	2547
	× LUMOS	**1.73×**	**1.69×**	**1.70×**
LP	GridGraph	1054	1421	4583
	LUMOS-BASE	805	935	2976
	LUMOS	624	826	2728
	× LUMOS	**1.69×**	**1.72×**	**1.68×**

Table 5: Execution times (in seconds) for LUMOS, LUMOS-BASE and GridGraph. Bold numbers indicate speedups of LUMOS over GridGraph.



(a) Throughput. (b) I/O Wait.

Figure 5: I/O throughput and waiting times for PR on TT across 1, 2, 4, 8 and 16 threads.

ble 4. For experiments on h1.2xlarge, we use TT, FT and YH whereas RMAT29 is used for I/O scaling experiments. Since unweighted TT fits within 32 GB (on h1.2xlarge), we limit the memory of unweighted TT experiments to 16 GB.

We compare the following three versions:

1. **GridGraph (GG):** is the GridGraph system [38].
2. **LUMOS-BASE (LB):** is LUMOS based on Eq. 2. It performs cross-iteration propagation only across partitions.
3. **LUMOS (L):** is LUMOS based on Eq. 3. It also performs intra-partition cross-iteration propagation.

6.2 Performance

Table 5 shows the execution times for GridGraph, LUMOS-BASE and LUMOS on h1.2xlarge. We use Highest Out-Degree

Version	TT	FT	YH
LUMOS-BASE	75.6%	67.5%	88.3%
LUMOS	81.1%	72.6%	93.8%

Table 6: Percentage of cross-iteration propagations.

to In-Degree Ratio First strategy which enables high cross-iteration propagations as shown in Table 6 (we will evaluate different partitioning strategies in Section 6.4). As we can see, LUMOS-BASE and LUMOS accelerate GridGraph in all cases. LUMOS is 1.65-1.74× faster than GridGraph while LUMOS-BASE is 1.28-1.59× faster; this is due to the reduced amount of I/O performed which gets enabled via cross-iteration propagation. Figure 6 shows the time spent in reading partitions in each case normalized w.r.t. execution time of GridGraph. As expected in out-of-core graph processing, the execution time is dominated by disk reads; for HDD we observe that GridGraph typically spends 74-83% of the time in performing disk reads, while LUMOS spends only 27-41% of the time performing disk reads compared to GridGraph. This reduction in read times results from our cross-iteration propagation technique that eliminates repetitive I/O using secondary layouts.

To study disk utilization, we vary the number of threads and measure the disk throughput and wait latencies for LUMOS. Figure 5 shows the disk throughput and wait latencies for PR on TT across 1, 2, 4, 8 and 16 threads. As we can see, the utilization is high even when using a single thread and having more threads only helps to maintain the high utilization (230-240 MB/sec) whenever the utilization drops for single thread (red trenches in Figure 5). With more threads issuing more I/O requests and utilization remaining same, the cores essentially wait more for I/O requests to complete as threads increase for HDD (shown in Figure 5b). We also observe high wait latencies as threads increase in Figure 5b due to high number of I/O requests.

It is interesting to observe that a single thread is easily able to keep the disk busy (wait times 150-200 ms) as its measured sequential read bandwidth is 278 MB/sec. Furthermore, we observe a significant dip in waiting times between ∼60-90 seconds (shown in Figure 5b) which appear while processing secondary layout; these secondary layouts are smaller, and hence the I/O requests get served quickly.

6.3 I/O Scalability

We study the impact of scaling I/O on LUMOS by setting up a RAID-0 array of 2 to 4 HDDs on d2.4xlarge, and 2 to 4 SSDs on i3.8xlarge instance. The resulting read bandwidths are shown in Table 7.

	Single Drive	RAID-0 with k drives		
		$k = 2$	$k = 3$	$k = 4$
d2.4xlarge (HDD)	195MB/s	368MB/s	590MB/s	768MB/s
i3.8xlarge (SSD)	1.2GB/s	3.8GB/s	4.1GB/s	3.9GB/s

Table 7: Sequential read bandwidth.

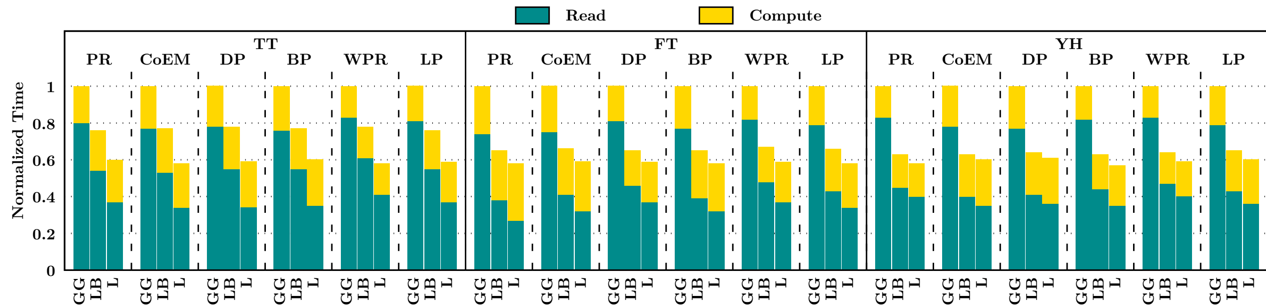


Figure 6: Read times and computation times of LUMOS and LUMOS-BASE normalized w.r.t. GridGraph's execution time.

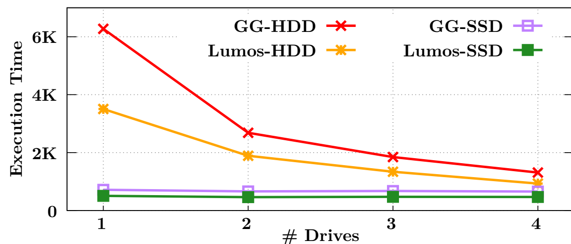


Figure 7: Execution times (in seconds) for LUMOS and GridGraph (GG) with varying number of drives. GG-HDD and LUMOS-HDD use HDDs on d2.4xlarge, while GG-SSD and LUMOS-SSD use SSDs on i3.8xlarge (see Table 7).

We ran PR on RAMT29 graph (8.6B edges) on the above setup with 64GB main memory. Figure 7 compares the execution times of LUMOS and GridGraph as number of drives increase from 1 to 4. LUMOS on d2.4xlarge scales gradually as I/O scales; it performs 1.8×, 2.6× and 3.8× faster with 2, 3 and 4 drives. It is interesting to observe that benefits of LUMOS over GridGraph diminish as number of drives increase. LUMOS performs 1.8× faster than GridGraph on a single drive and 1.3-1.4× faster on 2-4 drives. This is because GridGraph benefits from increased I/O bandwidth, which in turn leaves lesser room for effects of cross-iteration value propagation to become visible.

Contrary to HDDs, performance of LUMOS and GridGraph doesn't vary much as SSDs increase. Going from a single SSD to 2 SSDs reduces LUMOS's execution time from 505 sec to 460 sec, and the benefits of LUMOS over GridGraph also remain low with more SSDs (1.4×). This is again due to the high bandwidth provided by SSDs on i3.8xlarge (see Table 7) that alleviate I/O bottlenecks.

6.4 Partitioning Strategies

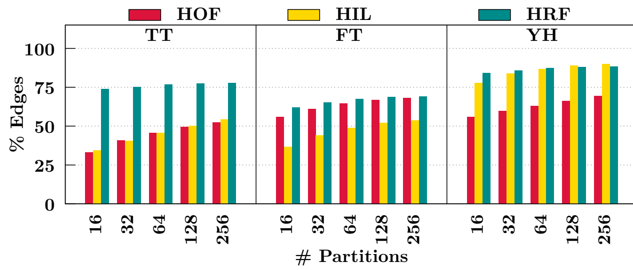
We evaluate our three light-weight partitioning strategies proposed in Section 5.3: Highest Out-Degree First (HOF), Highest In-Degree Last (HIL) and Highest Out-Degree to In-Degree Ration First (HRF). In Figure 8, we measure the amount of cross-iteration propagation for each of these strategies with and without intra-partition propagation and study sensitivity of these strategies to partition sizes.

While cross-iteration propagation depends on the structure of graph, HOF and HIL achieve 13-89% propagations across partitions whereas HRF captures the best of both and achieves significantly higher propagations (51-88%). With intra-partition propagations, all three strategies achieve significantly higher cross-iteration propagation (up to 92% for TT, 96% for FT, and 97% for YH). It is interesting to note that HOF slightly outperforms HRF and HIL for FT while HIL slightly outperforms HRF and HOF for TT; nevertheless, HRF remains useful since it achieves the middle ground between out-degree and in-degree metrics.

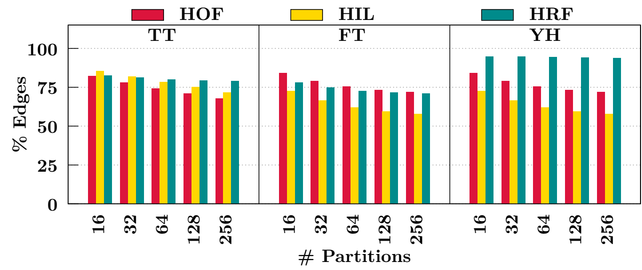
Finally, we observe that cross-iteration propagations across partitions increase as number of partitions increase and partitions become smaller; this is expected since edges within the same partition become potential candidates for propagations as they move to other partitions when partitions become smaller. Furthermore, when intra-partition propagation is enabled, there are fewer candidates within the same partition as number of partitions increase and hence, we observe a decreasing trend of cross-propagations. However, the scale of cross-iteration propagations remains high for HRF (70-97%) across different partition sizes, making it effective in all cases.

6.5 Preprocessing

Figure 9a shows the preprocessing times normalized w.r.t. GridGraph. While our light-weight partitioning strategy requires an additional pass over edges to compute vertex degrees, the pass is lightweight since it doesn't incur simultaneous writing of edges. Furthermore, edges don't need to be sorted and vertices are ordered across buckets that determine partitions. Finally, since majority of edges enable cross-iteration propagation, secondary layouts are smaller and hence, writing them out on disk is less time consuming than that for the original graph. Figure 9b shows the increase in disk space normalized w.r.t. GridGraph; as expected, the increase is only 12-33% for LUMOS-BASE because 67-88% of edges participate in cross-iteration propagation and hence, only remainder edges are present in the secondary layouts. With intra-partition propagation, the disk space requirement increases only by 7-26%.

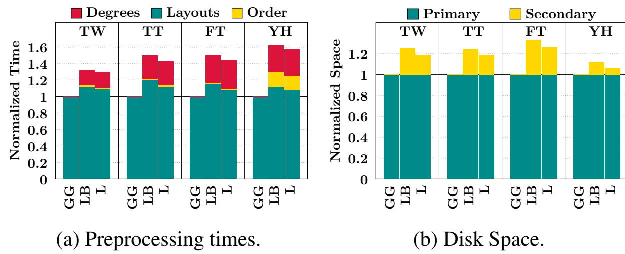


(a) Without locality-aware intra-partition propagation.



(b) With locality-aware intra-partition propagation.

Figure 8: Cross-iteration propagation enabled by three partitioning strategies: Highest Out-Degree First (HOF), Highest In-Degree Last (HIL), and Highest Out-Degree to In-Degree Ratio First (HRF).



(a) Preprocessing times.

(b) Disk Space.

Figure 9: Preprocessing time and disk space for LUMOS and LUMOS-BASE normalized w.r.t. GridGraph whose absolute time/space numbers are: TW = 217s/11GB; TT = 248s/15GB; FT = 334s/20GB; YH = 815s/50GB.

7 Related Work

We classify out-of-core graph processing systems into two categories based on the guarantees they provide.

(A) Synchronous Out-of-Core Graph Processing.
We discussed GridGraph [38] in Section 2; here, we briefly discuss the remaining works. GraphChi [13] pioneered single machine based out-of-core graph processing by designing partitions called *shards*, and developing a parallel sliding window model to process shards such that random disk I/O gets minimized. X-Stream [24] performs edge-centric processing using scatter-gather model. To reduce random vertex accesses, X-Stream partitions vertices and accesses edge list and update list based on partitioned vertex sets. Chaos [23] scales out X-Stream on multiple machines. FlashGraph [35] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. TurboGraph [8] is an out-of-core computation engine for graph database based on sparse matrix-vector multiplication model. Mosaic [17], GraFBoost [9] and Garaph [16] perform out-of-core processing on heterogeneous architecture containing high-bandwidth NVMe SSDs, massively parallel Xeon Phi processors, FPGAs and GPUs. Graphene [15] uses an I/O request centric graph processing model to simplify IO management by translating high-level data accesses to fine-grained IO requests. DynamicShards [31] develops dynamic partitions that eliminate unnecessary edges from partitions to reduce disk I/O.

Limitation: Since all of these works focus on computations within a single iteration, none of them leverage cross-iteration value propagation as LUMOS does. Furthermore, since these systems effectively process in partition-by-partition fashion, they can be further improved using LUMOS.

(B) Asynchronous Out-of-Core Graph Processing.
CLIP [1] exploits algorithmic asynchrony by making multiple passes over the partitions in memory. Wonderland [34] extracts effective graph abstractions to capture certain graph properties, and then performs abstraction-guided processing to infer better priority processing order and faster information propagation across graph. While the abstraction-based technique is powerful, its scope of applications is limited to path-based monotonic graph algorithms beyond which its applicability remains undefined (as mentioned in [34]). AsyncStripe [4] uses asymmetric partitioning & adaptive stripe-based access strategy to process asynchronous algorithms.
Limitation: Since synchronous guarantees are not provided by these works, their applicability is limited to asynchronous path-based algorithms. LUMOS with asynchronous processing semantics (Section 4) leverages relaxed dependencies for asynchronous algorithms as well.

Beyond Out-of-Core Graph Processing.
Google's Pregel [18], PowerGraph [6], GraphX [7], GPS [25] and Gemini [36] provide a synchronous processing model in a distributed environment, while Galois [20] and Ligra [26] offer similar guarantees in a shared memory setting. GraphBolt [19] provides synchronous processing semantics while processing streaming graphs.

8 Conclusion

We developed LUMOS, a dependency-driven out-of-core graph processing technique that performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees. Our evaluation showed that LUMOS computes future values across 71-97% of edges, hence reducing disk I/O and accelerating out-of-core graph processing by up to 1.8×.

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC*, pages 125–137, 2017.

[2] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW*, pages 595–602. ACM, 2004.

[3] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, pages 10–17, 2010.

[4] S. Cheng, G. Zhang, J. Shu, and W. Zheng. AsyncStripe: I/O Efficient Asynchronous Graph Computing on a Single Server. In *IEEE/ACM/IFIP CODES+ISSS*, page 32. ACM, 2016.

[5] Friendster network dataset. http://konect.uni-koblenz.de/networks/friendster. KONECT, 2015.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*, pages 17–30, Hollywood, CA, 2012. USENIX.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX OSDI*, pages 599–613, 2014.

[8] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *KDD*, pages 77–85, 2013.

[9] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *ISCA*. IEEE, 2018.

[10] U. Kang, D. Horng, and C. Faloutsos. Inference of Beliefs on Billion-scale Graphs. In *LDMTA*, 2010.

[11] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys*, pages 169–182. ACM, 2013.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *WWW*, pages 591–600. ACM, 2010.

[13] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX OSDI*, pages 31–46, 2012.

[14] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. MMap: Fast Billion-Scale Graph Computation on a PC via Memory Mapping. In *BigData*, pages 159–164, 2014.

[15] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *USENIX FAST*, pages 285–300, 2017.

[16] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. *USENIX ATC*, pages 195–207, 2017.

[17] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*, pages 527–543. ACM, 2017.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *ACM SIGMOD*, pages 135–146. ACM, 2010.

[19] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, pages 25:1–25:16. ACM, 2019.

[20] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *USENIX SOSP*, pages 456–471, 2013.

[21] K. Nigam and R. Ghani. Analyzing the Effectiveness and Applicability of Co-training. In *ACM CIKM*, pages 86–93. ACM, 2000.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[23] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *USENIX SOSP*, pages 410–424, 2015.

[24] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *USENIX SOSP*, pages 472–488, 2013.

[25] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, page 22. ACM, 2013.

[26] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN PPoPP*, pages 135–146, 2013.

[27] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] K. Vora. *Exploiting Asynchrony for Performance and Fault Tolerance in Distributed Graph Processing*. PhD thesis, University of California, Riverside, 2017.

[29] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*, pages 237–251, 2017.

[30] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *OOPSLA*, pages 861–878, 2014.

[31] K. Vora, G. H. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, pages 507–522, 2016.

[32] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Programmable and Budget-Aware Analytical Queries over Very Large Graphs on a Single PC. In *USENIX ATC*, pages 387–401, 2015.

[33] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[34] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*, pages 608–621. ACM, 2018.

[35] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *USENIX FAST*, pages 45–58, 2015.

[36] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX OSDI*, pages 301–316, 2016.

[37] X. Zhu and Z. Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. 2002.

[38] X. Zhu, W. Han, and W. Chen. GridGraph: Large Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*, pages 375–386, 2015.

NeuGraph: Parallel Deep Neural Network Computation on Large Graphs

Lingxiao Ma[†*], Zhi Yang[†*]
Peking University

Youshan Miao
Microsoft Research

Jilong Xue
Microsoft Research

Ming Wu
Microsoft Research

Lidong Zhou
Microsoft Research

Yafei Dai
Peking University

Abstract

Recent deep learning models have moved beyond low dimensional regular grids such as image, video, and speech, to high-dimensional graph-structured data, such as social networks, e-commerce user-item graphs, and knowledge graphs. This evolution has led to large graph-based neural network models that go beyond what existing deep learning frameworks or graph computing systems are designed for. We present NeuGraph, a new framework that bridges the graph and dataflow models to support efficient and scalable parallel neural network computation on graphs. NeuGraph introduces graph computation optimizations into the management of data partitioning, scheduling, and parallelism in dataflow-based deep learning frameworks. Our evaluation shows that, on small graphs that can fit in a single GPU, NeuGraph outperforms state-of-the-art implementations by a significant margin, while scaling to large real-world graphs that none of the existing frameworks can handle directly with GPUs.

1 Introduction

Graphs are natural representations of many real-world data; examples include web graphs, social networks, e-commerce user-item graphs, and knowledge graphs. With a graph representation, graph-based learning tasks, such as vertex classification and link prediction, can be optimized effectively. There has been a recent surge of interest in extending neural network models to graph data [7, 8, 13, 17–19, 23, 25, 29, 37]. These methods, known as graph neural networks (GNNs), combine standard neural networks with iterative graph propagation: the property of a vertex is computed recursively (with neural networks) from the properties of its neighbor vertices.

However, neither the existing deep learning frameworks nor the existing graph systems could support GNN algorithms sufficiently. The lack of system support has seriously limited the ability to explore the full potentials of GNNs at scale. Deep learning (DL) frameworks such as TensorFlow [4], PyTorch [2], MXNet [12], and CNTK [50] are designed to express deep neural networks (DNNs) but do not naturally express and efficiently execute graph propagation models. Deep graph library (DGL) [1] supports programming GNNs by wrapping DL systems with a graph-oriented message-passing interface. While DGL addresses the expressiveness challenge, it does not yet explore deeply the opportunities to leverage graph-aware operations for efficient executions. Furthermore, none of these frameworks, including DGL, offer the needed scalability to handle large graphs: The highly connected nature of graphs means that graph propagation could easily involve a large portion of a large graph, especially for power-law or dense graphs. Processing even a single vertex requires that deep learning frameworks load a large amount of graph-related data (e.g., structure and feature data) into limited GPU memory.

With the vertex-program abstraction and graph-specific optimizations, existing graph processing systems [10, 15, 26, 28, 47] can naturally express iterative graph algorithms like PageRank and community detection, and scale them to graphs with billions of vertices and edges. But graph systems can hardly express neural networks (NNs) and lack key capabilities required by efficient DNN executions, such as the tensor abstraction, automatic differentiation and dataflow programming model.

We therefore advocate bridging deep learning systems and graph processing systems to enable a new framework for scalable GNN training. In this paper, we explore the design of a GNN processing framework on top of dataflow-based DL systems. We argue that by introducing the graph model to dataflow and recasting graph-specific optimizations as dataflow optimizations, we can enable the DL frameworks to support efficient and scalable DNN computation on graphs. To support this argument, we developed NeuGraph, an efficient GNN processing framework built on top of an existing dataflow engine.

[†] National Engineering Laboratory for Big Data Analysis and Applications, Center for Data Science, Peking University.

[*] Lingxiao Ma and Zhi Yang equally contributed to this work.

The work is done when Lingxiao Ma is an intern and Zhi Yang is a visiting researcher at Microsoft Research.

NeuGraph combines the dataflow abstraction with the vertex-program abstraction in a new programming model called SAGA-NN (Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks). SAGA can be considered as a variant of graph-parallel abstraction (e.g., GAS [26]). Unlike a traditional system where user-defined functions (UDFs) express vertex programs, UDFs in SAGA-NN express NN computation on tensors as vertex or edge data, e.g., vertex or edge data. With the new programming model, NeuGraph allows users to express a GNN algorithm without worrying about the underlying system implementation (e.g., GPU memory management or scheduling). The graph-aware dataflow engine in NeuGraph judiciously partitions the graph data (vertex and edge data) into *chunks* (subgraphs), constructs the dataflow that operates at the chunk granularity, and schedules parallel executions of the dataflow on GPUs.

Naively adapting optimizations developed in the context of graph processing systems can lead to inefficient dataflow executions on DL frameworks. NeuGraph achieves high efficiency by introducing a range of optimizations both in the scheduling of parallel chunk processing, as well as the execution of core graph propagation procedures (i.e., Scatter-ApplyEdge-Gather stages) over the often-sparse graph structure. With fine-grained graph partitioning, NeuGraph achieves efficient *selective scheduling* and *pipeline scheduling* on top of the dataflow, to hide data movement between GPU and host when scaling a model out of the GPU core. To continue performance scaling, NeuGraph further adopts a new topology-aware scheduling strategy to efficiently distribute GNN models over modern multi-GPU systems. Finally, NeuGraph introduces computation-related optimizations for graph propagation, which is often hard to accelerate using GPUs.

We implemented NeuGraph on top of TensorFlow. We show that NeuGraph can support a variety of GNN algorithms on large graphs with millions of vertices and hundreds of millions of edges, as well as hundreds of feature dimensions over vertices, which existing DL frameworks cannot directly handle with GPUs. Compared on large graphs that TensorFlow can handle only with CPUs, NeuGraph achieves $16 \sim 47\times$ speedups. Even on small graphs that can fit into a GPU's memory, NeuGraph can still achieve a up to $5\times$ speedup over the state-of-art implementation on TensorFlow and a up to $19\times$ speedup over DGL [1]. Moreover, NeuGraph achieves nearly linear scalability over multiple GPUs.

As one of our key contributions, NeuGraph bridges two largely parallel threads of research, graph processing systems and dataflow-based DL frameworks, in the new GNN setting. NeuGraph significantly expands the capabilities of existing DL frameworks to support GNNs in the following key dimensions: programming model, graph partition and dataflow translation, graph propagation operations, and execution scheduling. We have also demonstrated, through extensive evaluation on real graphs with typical GNNs, significant benefits in scalability and efficiency by connecting graph processing and DL frameworks.

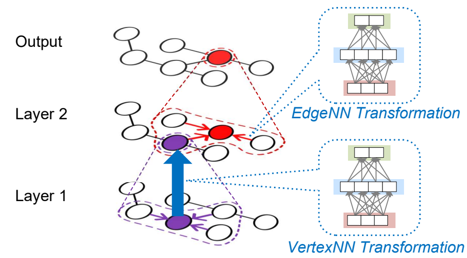


Figure 1: Feed-forward computation of a 2-layer GNN.

The rest of the paper is organized as follows. Section 2 introduces the SAGA-NN programming abstraction. Section 3 describes the optimizations in the NeuGraph system. Section 4 discusses the implementation and Section 5 presents our experimental results. We discuss related work in Section 6 and conclude in Section 7.

2 NeuGraph Programming Abstraction

In this section, we first reveal the essential structure of graph neural networks, and then propose our programming model that combines graph-parallel and dataflow abstractions.

2.1 Graph Neural Networks

Deep learning, in the form of deep neural networks, is a class of machine learning algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input. Deep learning has been gaining popularity due to its success in areas such as speech, vision, and natural language processing. In these areas, the coordinates of the underlying data representation often have a regular grid structure, which is friendly to hardware accelerators (e.g., GPU) with massive SIMD-style parallelisms.

Graph neural networks are deep learning based methods that operate neural networks on graph data, and have been adopted for many applications due to convincing in terms of model accuracy. Recently, several surveys [5, 46, 52, 54] provided a thorough review of different graph neural network models as well as a systematic taxonomy of the applications. A majority of GNN models can be categorized into *graph convolutional networks* [7, 9, 13, 19, 23], *graph recursive networks* [25, 33], and *graph attention networks* [43, 51].

We discuss 3 representative categories of GNNs with 3 representative models: (1) GCN [23] is a graph convolutional network that generalizes the notion of the convolution operation, typically for image datasets, and applies it to an arbitrary graph (e.g., a knowledge graph). GCN has been widely used in real-world scenarios like recommendation [6, 49]. Initially, each vertex in the graph has a feature vector. First, each vertex collects its neighbor vertices' feature vectors along edges, and

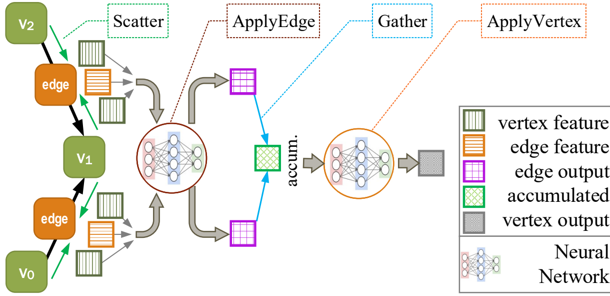


Figure 2: SAGA-NN stages for each layer of GNN.

sums the collected vectors (weighted by edge values). Then, a fully-connected NN is used to compute the vertex feature vector as the output. This is a layer of GCN. Stacking multiple GCN layers makes the vertex features representative enough for tasks. Taking the recommendation system as an example, a bipartite graph is constructed from the user-item ratings: There will be an edge with the rating as the edge value between the user vertex and the item vertex if a user rates an item. Then, the embeddings of both users and items can be learned by the GCN from the graph and the features of users and items. Finally, these embeddings are used to predict the missing user-item ratings to make a recommendation. (2) GG-NN [25] is a graph recursive network. It has an architecture similar to GCN, but uses different parameters for different edge types, as well as a Gated Recurrent Unit (GRU) in the NN to process accumulated features. (3) As a graph attention network, GAT [43] differs GCN mainly in that it computes an *attention* value for each edge during transferring vertex features.

In general, these GNN models share the same basic idea of collectively aggregating information following the graph structure. Specifically, each vertex or edge in the graph can be associated with a set of tensor data (normally a vector) as its feature or embedding. A GNN can consist of multiple layers, with an iterative propagation procedure conducted layer-by-layer over the same graph, as illustrated in Figure 1. At each layer, the vertex or edge features are transformed and propagated along edges, and then aggregated at the target vertices to produce new features for the next layer. Different from traditional graph algorithms (e.g., PageRank), the transformation on either vertices or edges can be arbitrary DNN computation. The GNN may also contain a label for each vertex, each edge, or the entire graph, for computing a loss function at the top layer. A feed-forward computation is then performed from the bottom layer to the top, with back-propagation conducted reversely.

Comparing with DNNs, the complexity due to graphs in GNNs creates a significant scalability challenge. First, real-world graphs, such as social networks or e-commerce networks, can easily have millions of nodes and edges. Second, vertices and edges in the graph are interconnected and need to be modeled as a whole neural network (i.e., a large, sparse neural network architecture defined according to a graph structure). This is particularly challenging on GPUs given the limited GPU memory capacity. Finally, unlike image, audio, or text that have clear grid structures, graph data are irregular, making it hard to conduct parallel GNN computation efficiently on GPUs.

2.2 A Running Example

We take the Gated Graph ConvNet (G-GCN) algorithm [7,29] as a concrete running example (see Example 2.1). G-GCN incorporates the gating mechanism into graph convolution. This model can be used to extract vertex features for community detection.

Example 2.1. *Let $\mathbf{h}_u^\ell$ denote the feature vector of a vertex u at layer ℓ, and W^ℓ, W_H^ℓ, and W_C^ℓ be the weight parameters to learn. G-GCN recursively defines the feature of a vertex u as follows:*

$$\mathbf{h}_u^{\ell+1} = ReLU\left(W^\ell \otimes \left(\sum_{v \to u} \eta_{vu} \odot \mathbf{h}_v^\ell\right)\right) \tag{1}$$

where $\otimes$ refers to matrix multiplication, $\odot$ refers to element-wise multiplication, and η_{vu} (for each edge $v \to u$) acts as edge gate,

$$\eta_{vu} = sigmoid\left(W_H^\ell \otimes \mathbf{h}_u^\ell + W_C^\ell \otimes \mathbf{h}_v^\ell\right) \tag{2}$$

where ReLU and sigmoid are nonlinear activation functions in neural networks.

G-GCN can be mapped to the pattern of computing a layer in Figure 1: Equation 2 represents the EdgeNN to compute the edge weight. $\sum_{v \to u} \eta_{vu} \odot \mathbf{h}_v^\ell$ in Equation 1 collects features from neighbors, and ReLU $(W^\ell \otimes \cdots)$ in Equation 1 is the VertexNN to process the accumulated features.

2.3 SAGA-NN Model

Based on the common pattern observed in GNN models, we propose SAGA-NN (Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks) as a new programming model for GNNs. It combines dataflow and vertex-program to express the recursive parallel computation at a layer of a GNN. SAGA-NN splits the feed-forward computation into four stages: *Scatter*, *ApplyEdge*, *Gather*, and *ApplyVertex*, as illustrated in Figure 2.

SAGA-NN provides two user-defined functions (UDFs) for ApplyEdge and ApplyVertex respectively, for users to declare neural network computations on edges and vertices. The **ApplyEdge** function defines the computation on each edge, which takes `edge` and `p` as input, where `edge` refers to the edge data and `p` contains the learnable parameters of the GNN model. Each `edge` is a tuple of tensors [`src`, `dest`, `data`] representing the associated data of the source and destination vertices connected by the edge, as well as the edge associated data (e.g., edge weight). This function can be used to apply a

```
G-GCN(vertex^ℓ):  // computing vertex^{ℓ+1}
  params p = [W_H^ℓ W_C^ℓ W^ℓ]
  // Passing data over edges
  edge^ℓ=Scatter(vertex^ℓ)
  // edge-parallel computation
  acc = ApplyEdge(edge^ℓ, p):
    η = sigmoid(p.W_H^ℓ ⊗ edge^ℓ.dest + p.W_C^ℓ ⊗ edge^ℓ.src)
    return η ⊙ edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  vertex^{ℓ+1} = ApplyVertex(vertex^ℓ, accum, p):
    return ReLU(p.W^ℓ ⊗ accum)
  return vertex^{ℓ+1}
```

Figure 3: Gated Graph ConvNet at layer ℓ in *SAGA-NN* model.

neural network model on edge and p, and outputs an intermediate tensor data associated with the edge. The **ApplyVertex** function defines the computation on a vertex, which takes as input a vertex tensor data vertex, the vertex aggregation accum, and learnable parameters p, and returns the new vertex data after applying a neural network model. The SAGA-NN abstraction builds on a dataflow framework, so users can symbolically define the dataflow graphs in UDFs by connecting mathematical operations (e.g., *add*, *tanh*, *sigmoid*, *matmul*) provided by the underlying framework.

The other two stages, **Scatter** and **Gather**, perform data propagation and prepare data collections to be fed to ApplyEdge and ApplyVertex as input. They are triggered and conducted by the system implicitly. We chose not to expose UDFs for Scatter and Gather, because these functions, if provided, are highly coupled with the propagation procedure, whose computations flow through the irregular graph structure and are difficult to express as dataflow that NeuGraph optimizes—users would have to implement the corresponding derivative functions of the UDFs, a serious burden. Following the same principle, NeuGraph also avoids exposing user-defined aggregation methods. It provides a set of default ones instead, including *sum*, *max* (e.g., *max-pooling* operator [18]), and concatenation, which can be chosen by setting **Gather**.accumulator.

NeuGraph models a GNN as a sequence of SAGA stages. The *Scatter* passes the vertex data vertex onto its adjacent edges to construct edge data edge, including both the source and destination vertex data. The subsequent *ApplyEdge* then invokes a parallel computation defined by the UDF on the edge data to produce an intermediate tensor value for each edge as its outputs. The *Gather* then propagates those outputs along the edges and aggregates them at the destination vertices through commutative and associative accumulate operations. Finally, the *ApplyVertex* executes the computation defined in UDF on all vertices to produce updated vertex data for the next layer. The procedure in Figure 1 fits in the SAGA-NN model: The *ApplyEdge* and *ApplyVertex* represent the EdgeNN and VertexNN, respectively; the *Scatter* and *Gather* perform the propagation along edges. This mapping indicates that the GNNs following the procedure in Figure 1 could be implemented with SAGA-NN model, hence presents the generality of SAGA-NN.

Figure 3 illustrates the description of G-GCN (at layer l) in the SAGA-NN model. Scatter gives each edge $v \to u$ with vertices data $[\mathbf{h}_v^\ell, \mathbf{h}_u^\ell]$, and ApplyEdge computes per-edge update $\mathtt{acc}_{vu} = \eta_{vu} \odot \mathbf{h}_v^\ell = \mathtt{sigmoid}\left(W_H^\ell \otimes \mathbf{h}_u^\ell + W_C^\ell \otimes \mathbf{h}_v^\ell\right) \odot \mathbf{h}_v^\ell$. Next, Gather performs $\mathtt{accum}_u = \sum_{v:v\to u} \mathtt{acc}_{vu}$, and ApplyVertex computes $\mathbf{h}_u^{\ell+1} = \mathtt{ReLU}\left(W^\ell \otimes \mathtt{accum}\right)$.

The dataflow abstraction makes it easy to express neural network architectures and leverage auto-differentiation. With the dataflow abstraction in SAGA-NN, NeuGraph enjoys the flexibility of executing operations on vertices or edges in batch for increasing efficiency. The vertex-program in SAGA-NN allows users to express computations naturally by thinking like a vertex, and models common patterns in GNNs as well-defined stages, thereby enabling optimizing in both graph computation and dataflow scheduling.

3 NeuGraph System

NeuGraph provides a combination of the dataflow and vertex-program abstractions as the user interface. Under this abstraction, NeuGraph proposes graph-aware optimizations for GNN processing to achieve efficiency and scalability.

At a high level, NeuGraph consists of: 1) a translation engine that translates GNN expressed by the SAGA-NN model into a dataflow graph at chunk-granularity to enable GNN computation over large graphs in GPUs; 2) a streaming scheduler that minimizes data movement across the host and GPU memory and maximizes its overlap with computation. The scheduler also needs to be topology-aware for use of multiple GPUs; 3) a graph propagation engine for deep learning that employs a set of fast propagation kernels and fuses operations to remove redundant memory copies; 4) a dataflow execution runtime. NeuGraph requires no modifications to existing dataflow-based DL frameworks, offering a general method to combine graph and NN computation within existing DL frameworks. In this section, we focus on the first three design points as they are main contributions of NeuGraph.

3.1 Graph-Aware Dataflow Translation

Just as with DNNs, efficient use of GPUs is critical to the performance of GNNs, especially for large graphs. However, existing DL frameworks cannot handle large graphs directly on a GPU because graph data cannot fit into GPU memory.

To achieve scalability beyond the physical limitation of GPU memory, NeuGraph introduces graph-specific partitioning on top of the dataflow abstraction. Note that both vertex feature data and graph structure data can be large. NeuGraph thus applies a 2D graph partitioning: As illustrated in Figure 4, it slices vertex data into P equally-sized disjoint vertex chunks, and tiles the adjacency matrix (representing edges)

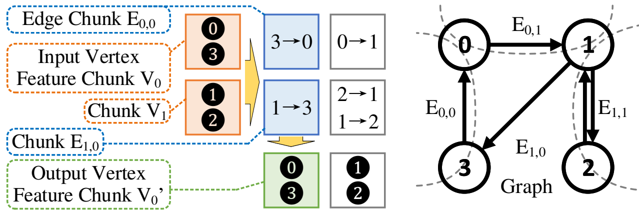


Figure 4: 2D Partitioning of a graph, here $P = 2$.

into $P \times P$ edge chunks. Edges in an edge chunk E_{ij} connect vertices in two vertex chunks V_i and V_j , respectively. By splitting graph data into chunks, NeuGraph can process edge chunks one by one, with only the source and destination vertex chunks needed for the edge chunk being processed. To achieve this, NeuGraph generates a dataflow graph with operators on data chunks, each of which fits in GPU memory, as illustrated in Figure 5.

For the forward computation at a layer, NeuGraph translates a dataflow subgraph for each destination vertex interval (e.g., a column in Figure 4): The Scatter operator inputs a specific edge chunk, i.e., the edge chunk in the i-th row and j-th column, and the associated i-th and j-th vertex chunks, and outputs an edge data chunk containing tuples in the form of [`src`, `dest`, `data`]. Each edge data chunk can be processed by operators specified in the ApplyEdge UDF to produce another edge data chunk with the result data `acc` (as in Figure 3). The operators at the Gather stage accumulate each edge's data based on its destination vertex to generate the corresponding vertex accumulation data chunk. After the processing of all the edge chunks for a destination vertex interval is done, the operators specified in the ApplyVertex UDF process the vertex accumulation chunks and output new vertex data chunks for the next layer.

For back-propagation, as the UDFs for ApplyEdge and ApplyVertex are expressed as dataflow computations over regular tensors, NeuGraph can leverage auto-differentiation provided by the DL frameworks. Additionally, NeuGraph further provides the *backward*-Gather operator to distribute the accumulation gradient returned by the *backward*-ApplyVertex stage across edges, and the *backward*-Scatter operator to accumulate all the partial gradients returned by the *backward*-ApplyEdge stage for a vertex in the previous layer.

Note that it is not necessary to enforce strict global barriers between stages in the SAGA-NN model. NeuGraph can flexibly schedule the chunk-based operators simply based on the data dependencies described in the dataflow graph. The system maintains the working set of operators within GPU memory by employing explicit device-to-host (D2H) and host-to-device (H2D) operators to conduct data swapping between the host and GPU memory. Also, during a training process, some intermediate feature data (e.g., the result of matrix multiplication in the ApplyEdge stage as in Figure 5) relevant to vertex chunks or edge chunks will be used in back-

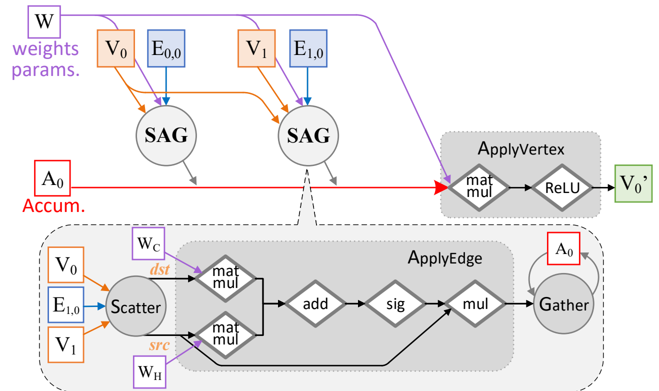


Figure 5: Chunk-based dataflow graph for a destination interval V_0 at a G-GCN layer. The backward dataflow graph and the swapping of intermediate results to host memory for backward are omitted for a clear visualization.

propagation. To save GPU memory, they are swapped out to host memory during the feed-forward computation and swapped back in during the back-propagation.

Discussion. The source vertex determines the row of the edge chunk and the destination vertex determines the column of the edge chunk. For every GNN layer, edge processing can be done in either a row-oriented or a column-oriented manner, based on the update pattern. For the forward computation, data flows from the source vertex to the destination vertex. With this pattern, row-oriented processing loses the opportunity of reusing the accumulated vertex data chunks, whose total size can be larger than the size of GPU memory. NeuGraph therefore adopts a column-oriented approach as illustrated in Figure 5, where it continuously executes operators in the Scatter-ApplyEdge-Gather (SAG) stages for V_0 and V_1 to produce A_0, which is subsequentially consumed by operators in the ApplyVertex stage. The destination vertex chunk and the corresponding accumulated vertex data chunk (e.g., A_0 in the figure) can be reused in GPU memory when NeuGraph processes edge chunks in the same column, so that data movement can be minimized.

By contrast, for the backward computation, a vertex gradient is propagated from the destination vertex to the source vertex. In this case, row-oriented processing is preferred. The vertex gradient data chunk can be reused from GPU memory when NeuGraph processes edge chunks in the same row. In the rest of this section, we focus on the discussion of the forward-pass execution of chunk-based dataflow, the backward-pass execution is done in a similar manner.

Besides the chunk processing order, determining the number of vertex chunks P is also important. Assuming edge chunks are accessed in the column-oriented manner in the forward pass, each edge chunk is accessed once, and each source vertex chunk is loaded P times. Thus, a smaller P is preferred to reduce I/O. NeuGraph selects P as the minimum integer to fit each chunk in GPU memory. Given a chunk-size choice

and the scheduling plan of the dataflow graph, NeuGraph computes the GPU memory requirement of the execution. If this requirement is beyond GPU's capacity, NeuGraph shrinks the chunk size by increasing P.

3.2 Streaming Processing out of GPU Core

For each layer, NeuGraph can scale GNN computation beyond the GPU core by processing the dataflow subgraph for a column of edge chunks (illustrated in Figure 5) in a column-by-column way. As we show later in the experiments (Table 2), the CPU-GPU data transfer has a significant impact on the overall performance, especially for sparse graphs. NeuGraph introduces a streaming scheduler with two innovations: selective scheduling that reduces data transfer on unnecessary vertices, and pipeline scheduling that maximizes the overlap between computation and data transfer.

Selective Scheduling. Unlike traditional graph algorithms (e.g., PageRank), the vertex data in GNNs can be much larger due to their high-dimensional feature vectors. To reduce the transfer cost of vertex chunks, NeuGraph exploits sparsity inherent in real-world graphs: To compute a specific edge chunk, not all vertices in the corresponding vertex chunks will be used due to the sparse graph structure (e.g., some vertices have no edges in this chunk). So, when processing an edge chunk E, NeuGraph applies a filter in CPU to select the useful vertices from E's source vertex chunk, and only transfers the selected vertex data into GPU.

We notice that a random graph partition (e.g., a permutation of the vertices) makes selective scheduling inefficient. Therefore, NeuGraph adopts a locality-aware graph partitioning algorithm (e.g., Kernighan-Lin algorithm) to condense as many edges that are connected to the same vertex as possible into one chunk (e.g., a diagonal one in the matrix of edge chunks). In this way, better access locality can be achieved for vertex data and hence more potential in selective scheduling.

Interestingly, when the majority of the vertices are useful (e.g., in a dense subgraph), directly transferring the full vertex chunk can be faster as it does not require additional memory copies for filtering. So for an edge chunk, we dynamically determine whether to apply the filtering in CPU based on the fraction θ of useful vertices. Given the host memory copy throughput T_{copy} on the CPU side, the filtering cost is $\frac{\theta}{T_{copy}}$. Let T_{trans} be the bulk transfer throughput from CPU to GPU. For a vertex chunk, if $\theta < \frac{T_{copy}}{T_{copy}+T_{trans}}$, NeuGraph chooses to apply filtering as it benefits the overall data transfer efficiency. Otherwise, NeuGraph skips the filtering and directly loads the entire vertex chunk into GPU.

Pipeline Scheduling. Besides the filtering optimization, NeuGraph further overlaps data transfer and computation through a pipeline scheduling to hide the transfer latency. Instead of streaming one edge chunk each time into GPU, NeuGraph can stream multiple chunks into the GPU device memory.

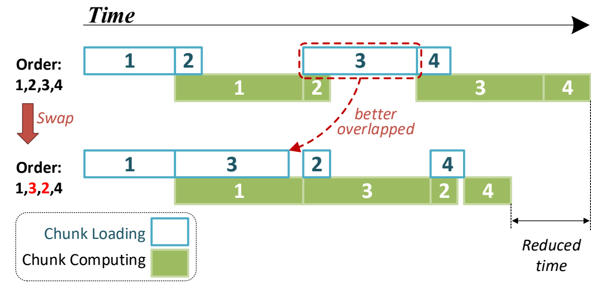


Figure 6: The swapping heuristic for a case of streaming two edge sub-chunks ($k = 2$).

In this case, a smaller chunk size can increase overlapping potential, which seems opposite to the requirement of a large chunk size to reduce vertex access I/O.

To deal with this dilemma, we apply the second-level partitioning over the edge grid to improve streaming efficiency without increasing the total I/O amount. Specifically, we horizontally partition an edge chunk and its associated source vertex chunk into k ($k \geq 2$) fine-grained sub-chunks, which enables parallel streaming processing of k sub-chunks. While performing computation on an edge sub-chunk, NeuGraph can simultaneously stream in other edge sub-chunks and their associated source vertex sub-chunks.

Recall that different edge sub-chunks could have distinct data transfer and computation cost due to different sparsity levels. NeuGraph carefully makes a scheduling plan for streaming heterogeneous sub-chunks. Given a column of edge sub-chunks, the system first generates the initial schedule plan by assigning a random order for processing. Next, it repeatedly swaps the order of a pair sub-chunks such that a better schedule plan with less time can be obtained. This process stops when it converges or reaches maximum iterations.

Then, NeuGraph exploits the cyclic pattern inherent in GNNs: Both the computation time and data transfer time of each sub-chunk can be profiled in the first several iterations and used in refining the scheduling plan for processing in the following iterations. Specifically, the system simulates the execution of the current schedule order based on the profiled execution information of individual sub-chunks. As illustrated in Figure 6, by examining the overlapping result in this simulation, the system finds a sub-chunk whose data transfer time is much shorter than the computation time, and within the same chunk, another sub-chunk is an opposite case. By swapping the order of these two heterogeneous edge sub-chunks, the system enables a better balance between the computation and data transfer.

3.3 Parallel Multi-GPU Processing

To improve scalability further, we can parallelize the training by partitioning the chunk-based dataflow (model parallelism) over multi-GPUs. Our dataflow graph is easy to parallelize due to its parallel nature, where GPUs can be assigned

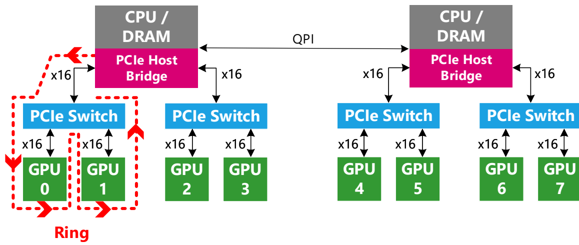


Figure 7: Multi-GPU architecture

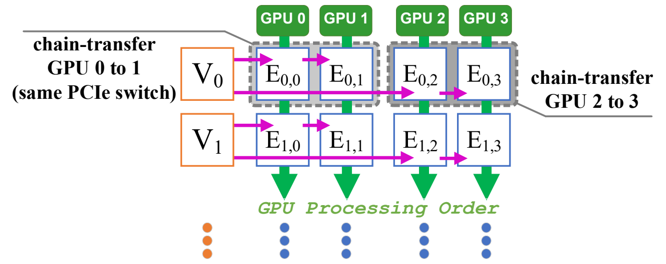


Figure 8: NeuGraph transfers vertex chunks along the chain.

dataflow subgraphs for different columns for cooperative processing.

However, with recent advances in hardware, modern multi-GPU systems introduce complex inter-connections among GPUs and across GPUs and CPUs, which presents new challenges to parallelize a dataflow graph. To illustrate this issue, Figure 7 shows the topology of a typical 8-GPU server, where GPUs are connected to CPU/DRAM (host memory) via a multi-level PCIe interface hierarchy. The upper level links that are shared by multiple communication paths can easily become a bottleneck. For example, GPUs 0 and 1 can only reach half of their peak bandwidth when reading edge/vertex data from host memory simultaneously, as limited by the link from the left-most PCIe switch to DRAM. Connecting the host to an accelerator like GPU via PCIe is the most common channel at present. We start from a common case, which may apply to other architectures.

To maximize the parallelism degree on multiple GPUs and prevent shared inter-connection links from becoming a bottleneck, NeuGraph employs a chain-based streaming scheduling scheme. Note that a vertex chunk is required by all the GPUs processing different columns of edge chunks. So, our idea is to let a GPU forward the vertex chunk (once loaded to its memory) to its neighbor GPU under the same PCIe switch, which can eliminate the bandwidth contention on the upper-level shared inter-connection link. NeuGraph therefore *logically* considers the GPUs under the same PCIe switch as a large *virtual GPU* and enables them to share data in a chain order as illustrated by the red dotted line in Figure 7.

In chain-based scheduling, each GPU streams one column of edge chunks and all vertex chunks to compute a destination vertex chunk. Note that the vertex data chunk for the destination interval can be initially loaded and cached in GPU memory. For simplicity, we assume that only the source vertex data is required for the computation. In particular, a GPU needs to take the following two operations: 1) loading an edge chunk from the host memory, and a data chunk from the host memory or from the device memory of its previous GPU in the chain, and 2) performing local computations. NeuGraph employs a coordinated scheduling to better overlap the two operations. As illustrated in Figure 8, we group GPUs into multiple virtual GPUs according to the inter-connection topology; e.g., GPUs 0 and 1 constitute one virtual GPU; GPUs 2 and 3 constitute another. Initially, GPUs 0 and 2 load vertex data chunk V_0 from the host memory. After loading, GPUs 0 and 2 start computing over chunk V_0, and also begin loading chunk V_1 from the host memory. Meanwhile, GPUs 1 and 3 start fetching chunk V_0 from GPUs 0 and 2, respectively. Next, GPUs 1 and 3 drop the data chunk V_0 after processing it locally as the chunk has already been consumed by all virtual GPUs. The whole process continues in such a pipelining fashion until all vertex data chunks have been loaded and processed.

In Section 3.2, we introduce the selective scheduling that can help reduce data movement between the host and GPU device memory. However, to apply selective scheduling in chain-based streaming, we need to select the useful vertex data required by the corresponding edge chunks in a virtual GPU; e.g., $E_{0,0}$ and $E_{0,1}$ in Figure 8. In a multi-GPU execution, we use the threshold $\theta = \frac{T_{copy}}{T_{copy}+T_{trans}}$ to determine whether or not to apply selective scheduling, where T_{copy} and T_{trans} are aggregative memory-copy and aggregative data-transfer throughput on both the CPU and GPU sides, respectively. Thus, given limited CPU resources shared by a large number of GPUs, NeuGraph applies selective scheduling on more sparse chunks with a larger θ.

3.4 Graph Propagation Engine

Besides ensuring high streaming efficiency, NeuGraph also introduces several important optimizations to reduce computation time in the execution of the Scatter-ApplyEdge-Gather (SAG) stages, which are not easily amenable to efficient GPU acceleration due to the often sparse edge structure of a graph.

First, NeuGraph incorporates a *dataflow graph optimization* to remove redundant computations in the SAG stage by considering the semantics of the SAGA-NN model. Consider the matrix multiplication operations in the ApplyEdge stage in Figure 5. These operations are conducted on vertex data that are scattered to a subset of edges and the learnable parameters W_C or W_H that are shared by all edges. Because a vertex may have multiple edges to which that the vertex data can be scattered, such a multiplication for a vertex can be conducted multiple times, leading to redundancies. NeuGraph therefore moves the computations that are related only to the source or destination vertices from the ApplyEdge stage of the current layer to the ApplyVertex stage of the previous layer.

Second, to support the Scatter and Gather stages efficient-

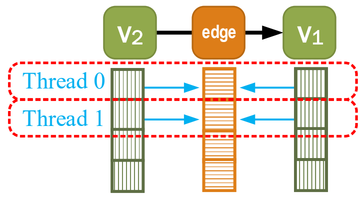


Figure 9: Parallelism along the dimension of feature vector.

ly on GPUs, NeuGraph provides *scatter/gather operation kernels* optimized for GPU executions. The design carefully considers the data structure layout to allow the kernel to better leverage the massive parallelism provided by GPU. In most GNNs, the data of each vertex is a dense vector rather than a scalar. We therefore exploit parallelism in per-vertex data access that fits better to GPU with SIMD architectures. Figure 9 illustrates the scatter kernel passing the vertex data, from both the source and the destination, onto an edge to form the edge data. We assign a thread block to process incoming edges with the same destination vertex. For vertices with a large in-degree, we divide the incoming edges into consecutive subgroups to be processed by multiple thread blocks. In a thread block, threads copy the source/destination vertex data into the edge data in parallel along the dimension of the vertex feature vector, ensuring good coalesced memory access. The gather kernel reduces the partial accumulation vectors `acc` from a set of edges that end at the same destination vertex `accum` into an accumulated vector. We employ a similar principle of exploiting parallelism for the scatter operator. A block of threads first cooperatively enumerate an edge group, accumulate the features of every edge into a temporary vector in GPU register, and finally write the result back to the corresponding destination vertex.

Finally, NeuGraph supports *Scatter-ApplyEdge-Gather (SAG) stage fusion* as another kernel optimization on execution of the propagation procedures. We find that, on most GNN applications, especially after the dataflow graph optimization, the ApplyEdge function only performs element-wise operations, such as +, -, ×, ÷, *tanh*, *sigmoid*, *ReLU*. In this case, we can optimize SAG stages by allowing the vertex/edge data to be directly updated with element-wise operations in GPU registers and then written back to their destination vertices in a single pass, without any extra cost of creating intermediate edge data in the GPU global memory. To achieve that, NeuGraph automatically detects such a case and replaces the whole SAG stages using a specially customized operation called *Fused-Gather*. This operation processes each edge chunk as follows: It first loads the inputs of Scatter; i.e., source vertices and edge data, into GPU registers, and then uses GPU threads to perform in-place updates directly on elements in registers based on user-defined element-wise operations in ApplyEdge. It finally produces the vector `acc`, which is summed onto the corresponding vertex accumulation vector `accum` with the user-defined Gather.accumulator.

4 Implementation

We implemented NeuGraph on top of TensorFlow (v1.7) with about 5,000 lines of C++ code and 3,000 lines of Python code. NeuGraph uses TensorFlow as the dataflow execution runtime, and additionally provides three specialized modules for GNN applications: (1) an engine translating a vertex-centric symbolic program into dataflow; (2) a streaming scheduler implementing the core scheduling logic; (3) a graph propagation engine with optimized kernels for the proposed Gather/Scatter operators. We discuss several important aspects of our implementation next.

Dataflow Translation. NeuGraph provides a base class `GNNlayer` in addition to the conventional operators; users can easily define each layer of a GNN algorithm by providing a symbolic vertex-program. Then NeuGraph divides vertices and edges into chunks, and generates a chunk-based dataflow graph by appropriately connecting GNN-layers with Gather and Scatter according to the user program. NeuGraph preprocesses the graph using the min-cut partition of METIS [21], and organizes each edge chunk in the compressed sparse column (CSC) format for the feed-forward computation, while using the compressed sparse row (CSR) format for back-propagation computation.

Streaming Scheduler. To improve performance, the streaming scheduler first analyzes the received dataflow graph and incorporates the optimizations described in Section 3.2. NeuGraph implements a filtering operator running on the CPU side, and determines whether to apply it before the H2D operator of each vertex chunk based on the percentage of relevant vertices (i.e., selectivity). Also, NeuGraph profiles the transfer/computation information of edge chunks and revises the dataflow graph based on the refined scheduling plan discussed in Section 3.2.

Multi-GPU Execution. Different devices in NeuGraph need to communicate with one another for coordination. In existing DL frameworks, an operator is usually dispatched to a specific device, with its input and output tensors on the same device. The multi-GPU communication in NeuGraph is executed by a series of concurrent operators from different devices. In each operator, after memory is allocated on a device for communication, it will exchange addresses with other devices for upcoming device-to-device data transfer. Parameters in different GPUs also need synchronization in each iteration. This is implemented by all-reduce.

Graph Propagation Engine. The graph engine contains graph-specific operator kernels. NeuGraph has optimized implementations for the proposed operators (*gather*, *scatter*, *fused-gather*). Specifically, *scatter* is a map operator that turns vertex data into edge data, and *gather* is a reduce operator that accumulates edge data for each vertex. Also, NeuGraph implements *fused-gather* operator described in Section 3.4 to enable one-pass edge computation when the edge computa-

```
CommNet(v^ℓ): // computing v^{ℓ+1}
  params p = [W_H^ℓ, W_C^ℓ]
  // Passing data over edges
  edge^ℓ = Scatter(v^ℓ)
  // no edge-parallel computation
  acc = ApplyEdge(edge^ℓ):
    return edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  v^{ℓ+1} = ApplyVertex(v^ℓ, accum, p):
    return ReLU(p.W_H^ℓ ⊗ v^ℓ + p.W_C^ℓ ⊗ accum)
  return v^{ℓ+1}
```

Figure 10: CommNet in SAGA-NN

```
GCN(v^ℓ): // computing v^{ℓ+1}
  params p = W^ℓ
  // Passing data over edges
  edge^ℓ = Scatter(v^ℓ)
  // edge.data is static weight
  acc = ApplyEdge(edge^ℓ):
    return edge^ℓ.src × edge^ℓ.data
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  v^{ℓ+1} = ApplyVertex(v^ℓ,accum,p):
    return ReLU(p.W^ℓ ⊗ accum)
  return v^{ℓ+1}
```

Figure 11: GCN in SAGA-NN

```
GG-NN(v^ℓ): // computing v^{ℓ+1}
  // different for each edge type
  params p, A
  edge^ℓ = Scatter(v^ℓ)
  // edge.data is edge type
  acc = ApplyEdge(edge^ℓ, A):
    return A(edge^ℓ.data) ⊗ edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data with GRU
  v^{ℓ+1} = ApplyVertex(v^ℓ, accum, p):
    return GRU(vertex^ℓ, accum)
  return v^{ℓ+1}
```

Figure 12: GG-NN in SAGA-NN

tion is element-wise.

5 Evaluation

In this section, we demonstrate the efficiency and scalability of NeuGraph by evaluating it on multiple GNNs and datasets.

GNN Models. NeuGraph can support many different types of graph-based neural networks [7, 8, 13, 18, 19, 23, 25, 29, 41]. We use the following three representative GNN models.

Communication neural network (*CommNet*) [41] is a model with which cooperating agents learn to communicate among themselves before taking actions. This network can be used to solve multiple learning communication tasks like traffic control. In CommNet, there is no computation on the edge, so the ApplyEdge stage is simply a passthrough (see Figure 10).

Graph convolutional network (*GCN*) [19, 23] applies convolutional operations to an arbitrary graph, and has been used in many semi-supervised or unsupervised graph clustering problems, such as entity classification in a knowledge graph. GCN (see Figure 11) has a computation (without neural networks) on the edge for weighted neighbor activation.

Gated graph sequence neural network (*GG-NN*) [25] applies recurrent neural networks (RNNs) to graph data and is used for NLP tasks. GG-NN performs NN-based edge computation (see Figure 12), with different parameters for different edge types. It also performs a heavy Gated Recurrent Unit (GRU) computation on vertices.

We chose these GNNs as the benchmark algorithms in the evaluation not only because of their different computation patterns, but also for the purpose of comparing with TensorFlow: the propagation stage in these cases can be treated as a sparse matrix multiplication and therefore expressible in TensorFlow. Certain algorithms such as G-GCN in our running example cannot be directly supported using the TensorFlow multiplication operators.

Datasets. Table 1 lists the real-world datasets used for evaluation, including the PubMed citation network (pubmed) [38], the BlogCatalog social network (blog) [42], the Reddit online discussion forum (reddit-small, reddit-full) [18], the Wikipedia data dump (enwiki) [3], and the Amazon data dump

Dataset	vertex#	edge#	feature	label	avg. degree
pubmed	19.7K	108.4K	500	3	5
blog	10.3K	668.0K	128	39	65
reddit-small	58.2K	1.4M	300	41	25
reddit-full	2.4M	705.9M	300	50	292
enwiki	3.6M	276.1M	300	12	77
amazon	8.6M	231.6M	96	22	27

Table 1: Datasets (K: thousand, M: million).

(amazon) [30]. The column *feature* in Table 1 reports the sizes of the vertex feature vectors, and the *label* column contains the numbers of label classes. As different GNN tasks share the same GNN architecture and differ only on the output layer, we tested the performance of our system on the task of vertex classification (e.g., classifying academic papers into different subjects in the PubMed citation dataset, which contains sparse bag-of-words feature vectors for each document and a list of citation links between documents) and set the number of layers $\ell = 2$ in experiments.

Environment and Baselines. We evaluated NeuGraph on a multi-GPU server, which is equipped with dual 2.6 GHz Intel Xeon E5-2690v4 processors (28 cores in total), 512 GB memory, and 8 NVIDIA Tesla P100 GPUs. The installed operating system is Ubuntu 16.04, using the libraries CUDA 9.0 and cuDNN 7.0.

We compared NeuGraph (NG) with TensorFlow v1.7 (TF) [4], GraphSAGE [18] (TensorFlow backend) and DGL v0.1.3 (PyTorch v1.0 [2] backend) [1]. GraphSAGE is a modeling framework for inductive representation learning on graphs and is widely used to generate low-dimensional vector representations for vertices. DGL is a Python package that serves as an interface between any existing tensor libraries and data expressed as graphs, thereby making it easy to implement GNNs.

We took the existing open-source implementations [1, 18, 23] [i]. We also implemented a basic extension, integrating TensorFlow with the chunk-based dataflow translation (TF-SAGA). The TF-SAGA can support larger GNN models, but

[i] For fair comparison, we took minor optimizations (e.g., replacing inefficient *feed_dict* with preloaded data tensors in memory to avoid redundant memory copies from python runtime to TensorFlow runtime).

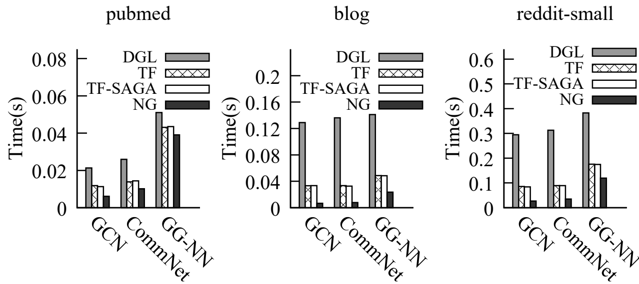


Figure 13: End-to-end performance comparison among DGL, TensorFlow (TF), TF-SAGA and NeuGraph (NG) on small datasets. GraphSAGE runs OOM.

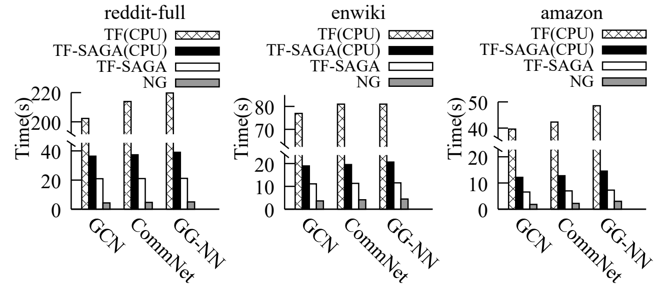


Figure 14: NeuGraph end-to-end performance comparisons on different large datasets. TensorFlow uses CPU-only mode as OOM occurs on GPU. TF-SAGA (CPU) is configured to run on CPU only, whereas TF-SAGA is GPU-enabled.

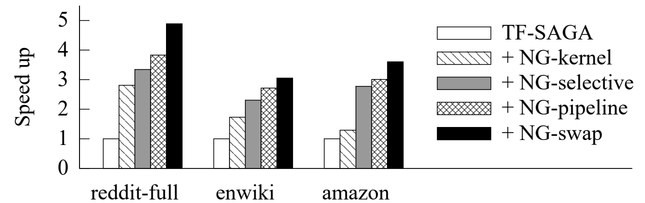


Figure 15: NeuGraph performance improvement breakdown of end-to-end on GCN model over different large datasets. The speedup is measured over the TF-SAGA (speedup = 1).

with all other optimizations described in Section 3 disabled. The comparison with TF-SAGA can reveal how much each optimization contributes to the overall performance.

We focused on metrics for system performance; e.g., time to scan one epoch of data. NeuGraph produces the same numerical results as TensorFlow and DGL, and hence has the same per-epoch convergence. All performance numbers in our experiments are calculated by computing the averages over 10 epochs.

5.1 Performance on a Single GPU

First, we evaluated NeuGraph by comparing it with the state-of-the-art frameworks TensorFlow, DGL, and GraphSAGE. As TensorFlow and DGL can only process graphs that fit in the device memory of a single GPU, we conducted these experiments on the first three small graphs in Table 1.

Figure 13 shows the end-to-end comparison results among different models and datasets. Overall, NeuGraph achieves on average a 2.5× speedup (up to 5.0×) compared with TensorFlow, and on average an 8.1× speedup (up to 19.2×) compared with DGL. We found that the properties of both graphs and models impact performance. NeuGraph achieves the largest speedup with GCN on the blog dataset. This is mainly because the high average vertex degree of the blog graph leads to greater graph propagation (i.e., SAG stages) costs, which NeuGraph can optimize more effectively.

Due to lack of graph support on TensorFlow, GraphSAGE implements GNNs through sampling neighbors and padding to convert irregular graphs to regular tensors. It leads to out of memory even on small graphs using the same evaluation setup (i.e., processing the whole graph with the sampler disabled). Moreover, it still runs about 5× slower than NeuGraph for GCN on pubmed even if the sampler is set to sample exactly one neighbor per vertex.

5.2 Scaling-up on a Single GPU

Since TensorFlow failed to process large graphs on GPU due to the out of memory (OOM) exceptions, we ran TensorFlow only on CPU. Accordingly, besides running TF-SAGA on GPU, we also ran it on CPU. DGL also experienced OOM exceptions when directly processing large graphs on GPU, therefore requiring additional graph sampling to alleviate memory pressure at the expense of model capacity and convergence guarantee. By contrast, NeuGraph can scale GNNs beyond GPU memory without loss on the model scale. Note that NeuGraph can also support the same graph sampling approaches as DGL. In this case, the results in Section 5.1 have already demonstrated that NeuGraph significantly outperforms DGL for small model scales on a single GPU. Hence, we do not compare them again here but focus instead on model scales that cannot fit in GPU memory.

End-to-end Comparison. Figure 14 shows the end-to-end comparison results among different models and datasets. Under the same CPU-only mode, TF-SAGA can achieve on average a 4.3× speedup over TensorFlow. That is because TF-SAGA on CPU contains finer grained chunk-level operators, which can be processed concurrently on the CPUs and make better use of the CPU resources. Moreover, NeuGraph achieves 16 $\sim$ 47× speedups compared to TensorFlow-CPU, which is the current solution for large graphs.

Compared with TF-SAGA on GPU, NeuGraph could provide even better performance with its additional optimizations. Figure 14 shows that NeuGraph achieves 2.4 $\sim$ 4.9× speedups over the GPU-enabled TF-SAGA on different models and datasets. Similar to those on small graphs, the speedups on large graphs depend on the graph structure. The average speedup across all models on the reddit-full graph with the highest vertex degree is 4.6× over the GPU-enabled

Time (s)	TF-SAGA			NeuGraph		
Dataset	IO	Comp.	Runtime	IO	Comp.	Runtime
reddit-full	7.67	13.27	20.94	3.84	2.46	4.28
enwiki	5.93	5.13	11.07	3.24	1.77	3.63
amazon	5.11	1.44	6.55	1.56	1.18	1.82

Table 2: GCN on large graphs: TF-SAGA vs. NeuGraph. NeuGraph overlaps I/O and computation time.

TF-SAGA, as opposed to 2.8× on the enwiki graph with moderate vertex degree and 3.1× for the amazon graph with the lowest vertex degree.

Breakdown Comparison. Both streaming and kernel optimizations can play important roles in achieving good overall performance after scaling GNN out of GPU core. To understand how much each optimization contributes to the overall performance, we disabled the graph propagation kernel optimization (NG-kernel) described in Section 3.4, as well as selective scheduling (NG-selective) and pipeline scheduling (NG-pipeline and NG-swap) described in Section 3.2. It effectively turns NeuGraph into the TF-SAGA. We then turned on these optimizations one by one and measured the resulting speedups they brought. To better understand the improvement, we also profiled the GCN execution on both TF-SAGA and NeuGraph with nvprof [32].

Figure 15 shows the improvement of each optimization over TF-SAGA for GCN. The results under other models are similar. We found that the graph kernel optimization works better on dense graphs (like reddit-full), whereas selective scheduling is more effective on sparse graphs (like amazon). For example, the graph kernel optimization can achieve a 2.8× speedup on the reddit-full graph, but only a 1.2× speedup on the amazon graph. However, selective scheduling can still bring an additional 2.6× speedup on the amazon graph. That is because a high-density graph leads to a higher computation cost on SAG stage, which is the target of the graph kernel optimization, whereas a low-density graph with selective scheduling can filter more unnecessary vertices. The figure also shows that our swap-based pipeline scheduling can bring significant improvement by effectively overlapping data transfer and computation, especially on the reddit-full graph where data chunks highly heterogeneous.

Table 2 shows the time of the host-device data transfer (I/O) and computation (Comp.) for TF-SAGA and NeuGraph. Compared to TF-SAGA, the optimizations in NeuGraph reduce both I/O and computation significantly and achieve good overlapping with pipeline scheduling.

As described in Section 3.1, the processing order of chunks may also impact performance. To examine the exact effect of processing order, we ran NeuGraph with the streaming processing optimizations described in Section 3.2 disabled. Figure 16 shows that, for the forward-backward pass, the column-row-oriented strategy is 1.4 ∼ 1.7× faster than the row-column-oriented one.

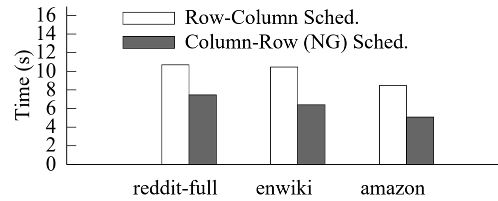


Figure 16: NeuGraph with row/column-oriented chunk scheduling: GCN on large graphs.

5.3 Scaling-out on Multiple GPUs

As described in Section 3.3, we can easily extend TF-SAGA from one GPU to multiple GPUs by allowing each GPU to process a dataflow subgraph, without considering the bandwidth contention. We compared it to NeuGraph with the chain-based scheduling disabled or enabled, in order to understand the performance of our topology-aware scheduling.

Figure 17 shows the results of the GCN model on three large graphs; the results of other GNN models are similar. NeuGraph significantly outperforms the multi-GPU TF-SAGA with the chain-based scheduling enabled or disabled. The average speedup of NeuGraph is 3.6×/2.7× over multi-GPU TF-SAGA with varying numbers of GPUs.

The benefit of the chain-based scheduling is highlighted in the comparison between enabling and disabling this topology-aware scheduling. For example, when scaling from 1 GPU to 2 GPUs, the average speedup of the disabled case even decreases, whereas the enabled one can improve from 3.8× to 5.5× over the single GPU TF-SAGA. This is mainly because, without the chain-based scheduling, two GPUs within the same PCIe switch need to load input edge/vertex data through a shared link concurrently, which can easily become the bottleneck. By contrast, the chain-based mechanism allows the second GPU to load vertex data directly from the first one, reducing the pressure on the shared PCIe link.

We observed that the chain-based scheduling achieves nearly linear speedup on the reddit-full and enwiki graphs, but exhibits less optimal results on the relatively sparse amazon graph. The reason is that NeuGraph tends to apply selective scheduling on relatively sparse graphs. However, given the limited CPU resources shared by an increasing number of GPUs, NeuGraph has to decrease usage of the CPU for per-GPU filtering. Also, the current TensorFlow implementation cannot support NUMA-aware tensor allocation well, which imposes a performance impact on the CPU filtering, especially on sparse graphs like the amazon where the filtering is often used.

6 Related Work

The growing scale and importance of graph data has driven the development of numerous specialized graph processing systems, including Pregel [28], GraphLab [26], Power-

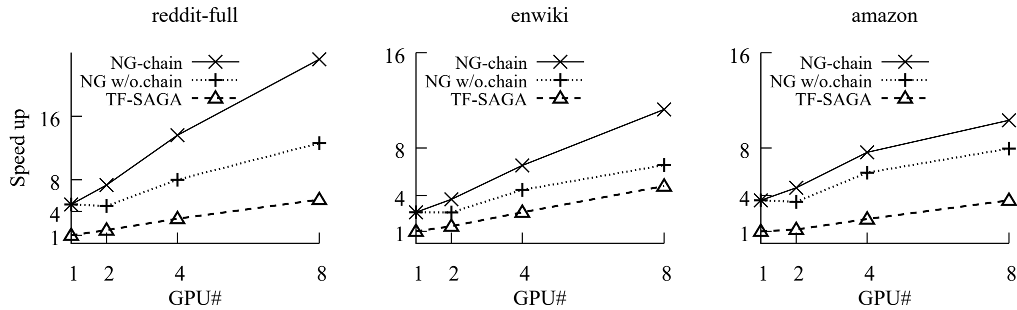


Figure 17: Scaling out GCN with NeuGraph on large graphs (w/o refers to without). The speedup is measured over the single GPU TF-SAGA (speedup = 1). Chain-base scheduling works on multi-GPU, resulting in the same 1 GPU point with it enabled/disabled.

Graph [15] and GraphX [16]. There are many other following works with optimizations on different aspects including graph layout, sequential data access, and secondary storage (e.g., GraphChi [24], Grace [34], FlashGraph [53], XStream [36] and Chaos [35]), distributed shared memory and RDMA (e.g., Grappa [31] and GraM [45]), NUMA-awareness, scheduling, and graph partitioning (e.g., PowerLyra [10] and Bi-Graph [11]). All these works focus on CPU based computation.

There is another series of system works that focus on exploiting GPU for large graph processing. GraphReduce [39] can process out-of-memory graphs on a single GPU and optimize memory coalescing by using two different formats. GTS [22] can also process out-of-memory graphs on multiple GPUs by fully exploiting the asynchronous GPU streams. Garaph [27] exploits edge-centric parallelism and dynamic scheduling to achieve the best performance on the CPU/GPU hybrid platform. Lux [20] investigates the placement of graph data over the CPU memory hierarchy on multiple nodes. All these graph processing systems are driven by basic graph benchmarks such as PageRank and shortest path, but lack the support for neural network computation, such as the tensor abstraction and auto-differentiation. To be compatible with existing DL libraries, NeuGraph chooses to recast the graph-specific optimizations as dataflow optimizations on top of DL frameworks (e.g., TensorFlow). This does not limit the capability of expressing a general DL computation, and allows users to benefit from both graph and DL optimizations.

TuX^2 [47] aims to bridge the gap between graph and traditional machine learning computation, while NeuGraph targets neural network computation on graphs, which connects graph processing and deep learning supported by the dataflow frameworks like TensorFlow [4], PyTorch [2], MXNet [12], and CNTK [50], etc. Most recently, Cavs [48] introduces the vertex-centric programming model into dynamic neural networks to address the problems that each sample has a unique dataflow graph and the training is iterative on batches of samples. NeuGraph addresses different problems and challenges regarding scalability and performance in supporting GNN models on large real-world graphs. DGL [1] wraps DL systems with a message-passing programming interface for GNNs, while NeuGraph addresses the system challenges (e.g., scalability and efficiency) by translating graph-aware computation on dataflow and recasting graph optimizations.

From the modeling perspective, there are several modeling works (e.g., GraphSAGE [18], MPNN [14], and GN-Block [5]) that attempt to unify existing GNNs into a single modeling framework. These generalized modeling frameworks can be implemented easily and executed efficiently at scale by NeuGraph. Recently developed graph sampling approaches (e.g., DGL [1], GraphSAGE [18], PinSAGE [49], and FastGCN [9]) alleviate scalability challenges of GNNs at the expense of model capacity and convergence guarantee. These approaches are orthogonal to and compatible with our work. NeuGraph frees users from choosing appropriate sample sizes and worrying about GPU memory limitations.

7 Conclusion and Future Work

GNN is an emerging computation model that arises naturally from the need to apply neural network models on graphs. Supporting efficient and scalable parallel computation for GNN training is demanding due to its inherent complexity. Given this new requirement, we advocate unifying graph computation and deep learning systems for GNNs. NeuGraph represents a critical step in this direction by showing not only the feasibility, but also the potential of such unification. We accomplish this by defining a new, flexible SAGA-NN model to express GNN algorithms by fusing graph-related optimizations into the management of data partitioning, scheduling and parallelism in deep learning frameworks.

One potential future direction is to scale GNN further to multiple servers, by leveraging the work in distributed graph systems [40, 44, 45].

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. We are particularly grateful to our shepherd Harry Xu for his detailed guidance in the final revision process.

References

[1] Deep graph library. https://github.com/dmlc/dgl, Retrieved January, 2019.

[2] PyTorch. http://pytorch.org, Retrieved January, 2019.

[3] Wikimedia downloads. https://dumps.wikimedia.org/, Retrieved May, 2018.

[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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 265–283. USENIX Association, 2016.

[5] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[7] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[8] Thang D. Bui, Sujith Ravi, and Vivek Ramavajjala. Neural graph learning: Training neural networks using graphs. In *Proceedings of 11th ACM International Conference on Web Search and Data Mining*, WSDM'18, pages 64–71. ACM, 2018.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, ICLR'18, 2018.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1:1–1:15. ACM, 2015.

[11] Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys'14, pages 14:1–14:7. ACM, 2014.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems*, LearningSys'16, 2016.

[13] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, NIPS'16, pages 3844–3852, 2016.

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, ICML'17, pages 1263–1272. JMLR. org, 2017.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 17–30. USENIX Association, 2012.

[16] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 599–613. USENIX Association, 2014.

[17] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, IJCNN'05, pages 729–734. IEEE, 2005.

[18] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, NIPS'17, pages 1024–1034, 2017.

[19] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

[20] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, November 2017.

[21] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs.

SIAM Journal on scientific Computing, 20(1):359–392, 1998.

[22] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, pages 447–461. ACM, 2016.

[23] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR'17, 2017.

[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 31–46. USENIX Association, 2012.

[25] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, ICLR'16, 2016.

[26] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[27] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 195–207. USENIX Association, 2017.

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 135–145. ACM, 2010.

[29] Diego Marcheggiani and Ivan Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, EMNLP'17, pages 1506–1515. Association for Computational Linguistics, 2017.

[30] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'15, pages 43–52. ACM, 2015.

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC'15, pages 291–305. USENIX Association, 2015.

[32] Nvidia Corporation. Profiler :: Cuda toolkit documentation. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`, Retrieved January, 2019.

[33] Nanyun Peng, Hoifung Poon, Chris Quirk, Kristina Toutanova, and Wen-tau Yih. Cross-sentence n-ary relation extraction with graph lstms. *Transactions of the Association for Computational Linguistics*, 5:101–115, 2017.

[34] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 41–52. USENIX Association, 2012.

[35] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 410–424. ACM, 2015.

[36] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 472–488. ACM, 2013.

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[38] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 20(1):61–80, 2008.

[39] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12. ACM, 2015.

[40] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 317–332. USENIX Association, 2016.

[41] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, NIPS'16, pages 2244–2252, 2016.

[42] Lei Tang and Huan Liu. Relational learning via latent social dimensions. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'09, pages 817–826. ACM, 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'18, 2018.

[44] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 651–664. USENIX Association, 2018.

[45] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC'15, pages 408–421. ACM, 2015.

[46] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[47] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. TuX2: Distributed graph computation for machine learning. In *Proceedings of the the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 669–682. USENIX Association, 2017.

[48] Shizhen Xu, Hao Zhang, 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 *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 937–950. USENIX Association, 2018.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'18, pages 974–983. ACM, 2018.

[50] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Technical Report MSR-TR-2014–112, October 2014.

[51] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. GaAN: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

[52] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *arXiv preprint arXiv:1812.04202*, 2018.

[53] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 45–58. USENIX Association, 2015.

[54] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines

Eunjae Lee
UNIST
kckjn97@unist.ac.kr

Junghyun Kim[*]
TmaxOS
junghyun_kim3@tmax.co.kr

Keunhak Lim[*]
NEXON Korea
limkeunhak@nexon.co.kr

Sam H. Noh
UNIST
next.unist.ac.kr

Jiwon Seo[†]
Hanyang Univ.
seojiwon@hanyang.ac.kr

Abstract

Many important graph algorithms are based on the breadth first search (BFS) approach, which builds itself on recursive vertex traversal. We classify algorithms that share this characteristic into what we call a BFS-like algorithm. In this work, we first analyze and study the I/O request patterns of BFS-like algorithms executed on disk-based graph engines. Our analysis exposes two shortcomings in executing BFS-like algorithms. First, we find that the use of the cache is ineffective. To make use of the cache more effectively, we propose an in-memory static cache, which we call BFS-Aware Static Cache or Basc, for short. Basc is static as its contents, which are edge lists of vertices that are pre-selected before algorithm execution, do not change throughout the execution of the algorithm. Second, we find that the state-of-the-art ordering method for graphs on disks is ineffective with BFS-like algorithms. Thus, based on an I/O cost model that estimates the performance based on the ordering of graphs, we develop an efficient graph ordering called *Neighborhood Ordering* or *Norder*. We provide extensive evaluations of Basc and Norder on two well-known graph engines using five real-world graphs including Twitter that has 1.9 billion edges. Our experimental results show that Basc and Norder, collectively have substantial performance impact.

1 Introduction

Algorithms such as breadth first search (BFS) [26], shortest paths (SP) [15], all pairs shortest path (APSP) [35], diameter computation (DIAM) [1], finding weakly connected components (WCC) [15], and betweenness centrality (BC) [4] are popular graph algorithms widely used in many domains including bioinformatics, social science, and economics. These algorithms share a commonality that they start from a given set of vertices and then recursively traverse their neighboring vertices. Together, we call algorithms with these characteristics BFS-like algorithms.

In BFS-like algorithms, only a subset of vertices are active at any given time. Furthermore, which of the vertices are to be activated among all the vertices is difficult to predict. Due to this reason, the locality of memory access in BFS-like algorithms is generally worse than that of other graph algorithms such as PageRank, where all vertices are active and regularly accessed [27].

Due to their poor locality of memory access, it is difficult to optimize the performance of BFS-like algorithms, particularly on disk-based graph engines that store the input graph on external storage such as SSDs. Although several optimization techniques have been suggested for disk-based graph systems, their impact on BFS-like algorithms is limited. Existing optimizations such as overlapping I/O and CPU operations [12,19] or merging small I/O requests into a single larger request [40] do not consider the characteristics of BFS-like algorithms hence, have substantial room for improvement [2,27].

The focus of this paper is on BFS-like algorithms, and our contribution can be summarized as follows. First, we present a thorough analysis of BFS-like algorithms running on disk-based graph engines. We observe and report characteristics not previously revealed such as the fact that the number of I/O requests for each vertex is similar among the vertices, regardless of their degrees or relative positions in graphs.

Second, based on our observations, we propose a new form of a cache, which we call Basc (an acronym for BFS-Aware Static Cache). Basc has three distinct characteristics as a cache: 1) it is separate space set aside from the typical page cache[1], 2) it holds edge lists of

[*]Participated in this research as graduate students at UNIST
[†]Corresponding author and principal investigator

[1]Without loss of generality, we will use the term ‘page cache’ to refer to typical caches that are deployed to improve I/O performance in graph engines or within operating systems.

certain pre-selected vertices, and 3) it is static, that is, the contents of the cache do not change throughout the execution of the algorithm. We show that by judiciously making use of Basc, performance of BFS-like algorithms can be substantially improved.

Finally, our observation shows that the performance of BFS-like algorithms is highly sensitive to the layout of the graphs. Based on this observation, we devise a simple model that estimates the I/O costs of BFS-like algorithms based on the layout of the graph on disk. We experimentally validate that the model is fairly accurate in estimating performance. Moreover, guided by the cost model, we develop a simple, yet efficient graph ordering scheme that we call *Neighborhood Ordering* or *Norder* for short, which substantially improves the performance of BFS-like algorithms, even while the time to compute the ordering takes substantially less than existing ordering schemes.

The methodologies that we propose are for disk-based graph systems adopting the vertex-centric computation model. As this model is widely adopted in large-scale graph analytics, our work is applicable to many existing graph processing systems [12, 21, 28, 34, 40]. For fair comparison with previous schemes we implement our methods in FlashGraph and Graphene, two recent graph engines [21, 40]. Note that all discussions hereafter are done in the context of disk-based graph systems.

The rest of the paper is organized as follows. Section 2 describes our analysis of BFS-like algorithms running on disk-based graph systems. Section 3 introduces Basc, our BFS-aware static cache, as well as the vertex selection algorithm that we propose. In Section 4, we develop our I/O cost model based on graph orderings, then propose an optimized graph ordering that we call Neighborhood Ordering. We evaluate our proposed techniques in Section 5 and discuss related work in Section 6. Finally, we end with conclusions in Section 7.

2 Characteristics of BFS-like Algorithms

In this section, we first discuss the basic workings of disk-based graph engines and BFS-like algorithms. We focus on semi-external graph engines that store vertex attributes in memory, as main memory of commodity computers today is typically large enough to hold vertex attributes in their entirety. Then, through Sections 2.2∼2.4, we discuss the characteristics of BFS-like algorithms that we observe in our analysis.

2.1 Basics of Disk-based Graph Engines

In vertex-centric computations, the entire set or a subset of vertices are activated as they receive messages in each iteration. Then the edge lists of the activated vertices are accessed when necessary to send messages to the neighboring vertices [25]. As they are accessed, these edge lists are retrieved from disk to memory in page granularity, whose size typically ranges from 1KB to a few MBs [7, 12, 34, 40]. These pages are then stored in the page cache, which is either controlled by the graph engine or the file system [7, 12, 19, 40].

The edge lists are generally sorted and indexed by the owner vertex ID and stored sequentially on disk. Some graph engines apply optimized partitioning schemes such as hybrid-cut partitioning, instead of the more general vertex-cut or edge-cut partitioning, to reduce I/O [6, 39, 43]. Also, when I/O requests are issued to retrieve the pages, requests for adjacent pages may be merged for higher throughput [21, 40].

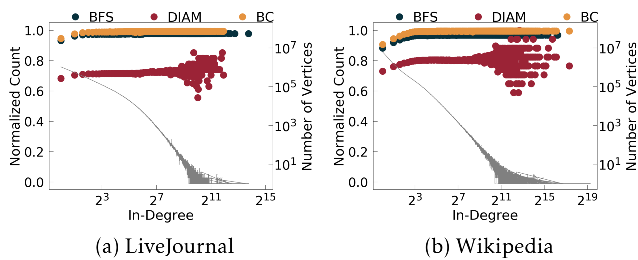
Performance of graph algorithms on disk-based graph engines depends largely on the efficiency of accessing the input graph [7, 19]. In disk-based engines, both the vertex attributes and input graphs are stored on disk that are randomly accessed by the graph algorithms. However, the overhead of accessing the input graph is generally higher than that of accessing vertex attributes as a large portion of vertex attributes are typically cached in memory. In particular, as mentioned previously, semi-external graph engines store all the vertex attributes in main memory.

Graph algorithms written in the vertex-centric model run iteratively, with a varying subset of vertices activated per iteration depending on the algorithm type. In BFS and BFS-like algorithms, only the "frontier" vertices are activated in each iteration. Thus, only the edge lists of these vertices are accessed in a random manner.

As the edge lists of activated vertices are accessed, pages containing these edges are loaded into the page cache. As page units are large, edges of unactivated vertices may also be in the retrieved page and hence, needlessly loaded to memory. To achieve high cache utilization, and consequently, high performance, we want the page cache to contain edges of as many activated vertices as possible. This requires the vertices in the input graph to be ordered such that the edges of activated vertices in the same iteration are stored in proximity.

2.2 Uniform Edge List Reference

Retrieving edge lists on disk has a significant effect on performance [19, 43]. To alleviate this burden, it would be desirable to cache the frequently requested edge lists of the vertices. To this end, we observe the edge list request pattern in representative BFS-like algorithms. Common logic tells us that for a vertex with a large edge list, that is, a large number of neighbor vertices, more requests will be targeted to that vertex and its edge list. However, interestingly and contrary to this



(a) LiveJournal (b) Wikipedia

Figure 1: Distribution of edge list reference counts: reference counts for vertices of the same in-degree are averaged and plotted relative to the vertex with the largest reference count for each algorithm. The gray line shows the number of vertices corresponding to each in-degree.

logic, we find that there are no substantial differences in the number of edge list accesses among the vertices, even for vertices that have widely varying degrees.

Figure 1, which are representative results, shows the distributions of the number of edge list requests for the BFS, DIAM, and BC algorithms with the LiveJournal and Wikipedia dataset [2]. The graphs show the average number of requests for the vertices with the same in-degrees, that is, having the same number of in-bound edges, normalized to the maximum average request count.

Initially, our conjecture was that the number of requests for a vertex (and thus, to its edge list) would be roughly proportional to its in-degree because messages (hence requests) are sent over edges in a vertex-centric model. However, Figure 1 shows that there is only a small difference in the number of edge requests between high and low in-degree vertices. The average counts are nearly constant and the variance (not shown) are low.

The reason for such uniform reference count is that in the vertex computation model, the edge list of a vertex is accessed only once per iteration as multiple requests to a vertex are merged into a single request if they are issued in the same iteration. Thus, for a high in-degree vertex whose neighbor vertices are densely connected to each other in real-world graphs [10, 20], the majority of requests to the vertex are sent as a single request. Moreover, in some BFS-like algorithms such as BFS, each vertex is processed only once in the entire running of the algorithm, with the exception of unreachable vertices. So the edge lists are also requested only once for each vertex. Thus, the number of edge list references is independent of the in-degrees and close to a constant.

The observation that there is no substantial difference in the number of references to the edge list tells us that strategies such as simply storing frequently accessed edge lists in memory is not an effective approach for improving performance. We take this observation to

[2]The full dataset descriptions are provided in Table 1.

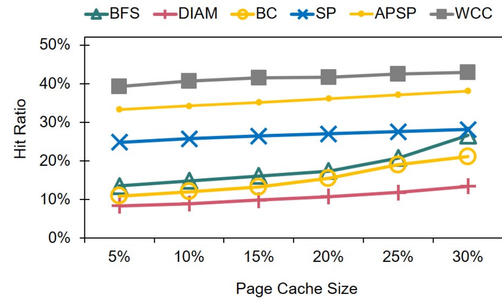


Figure 2: Page cache hit ratios for BFS-like algorithms. The page cache size is varied from 5% to 30% of the input graph size.

propose a different method for caching the edge lists of vertices, which we discuss in Section 3.

2.3 Ineffectiveness of the Page Cache

Locality per I/O access is known to be poor for BFS-like algorithms running on disk-based systems. This is because pages are brought into the page cache by active vertices and these active vertices are determined in a rather random manner at each iteration [23, 27]. To quantify this, we run experiments using the BFS-like algorithms provided in FlashGraph and observe the hit ratio of the page cache. Figure 2 depicts the results for page cache sizes ranging from 5% to 30% of the input graph run on the Twitter dataset. We see that all algorithms show low hit ratios. WCC, which shows the highest hit ratio, operates differently as the algorithm starts out with all the vertices, and as the component ID of each vertex converges, the number of active vertices decreases[3]. However, a small number of vertices linger around in later iterations, and those vertices fit in the page cache resulting in the higher hit ratio. More importantly, though, we find that for all algorithms, increasing the page cache has little impact on the hit ratio, improving only by 5 to 10% even with a six factor increase in page cache size.

Our conclusion here is that the page cache in disk-based graph engines does not play a major role in regards to performance for BFS-like algorithms. Simply increasing the page cache cannot be a solution, and there needs to be a different approach to resolve this ineffectiveness, for which Basc in Section 3 is proposed.

2.4 Impact of Graph Layout on Disk

Both SSDs and HDDs show faster performance with sequential reads than random reads [31]. Thus, how a graph is stored and accessed by the running algorithm has substantial impact on performance. In this section,

[3]Section 5 describes in detail how the algorithms are implemented.

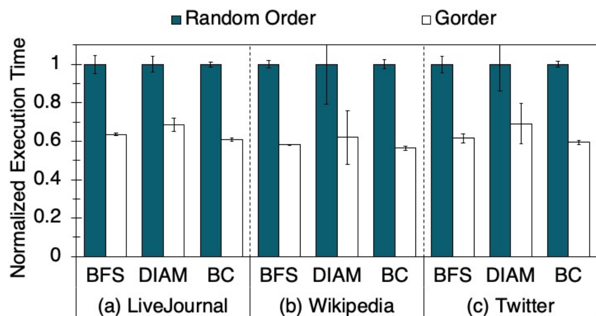


Figure 3: Performance of BFS, DIAM, and BC for Gorder normalized to random ordering for three datasets.

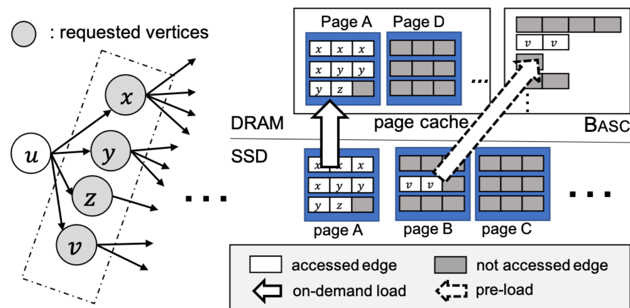


Figure 4: BASC and page cache interaction.

we perform experiments to help us understand the performance impact of graph layouts.

For the experiments, we restructure a graph in two different orderings and measure the performance of the graph algorithms. In the first, we randomly assign the vertex IDs, and in the second, we use Gorder [38], which was proposed to improve the locality of access to vertices and their edge lists for main memory graph systems. In particular, Gorder computes the ordering of vertices and their edge lists by optimizing its locality score, which is defined based on whether densely connected vertices are ordered closely within a given distance. Then, the graph is stored in CSR (Compressed Sparse Row) format, where the edge lists of vertices are ordered by their vertex IDs and stored sequentially.

Figure 3 compares the performance of three BFS-like algorithms with Gorder and random ordering on three datasets. We can see that the algorithms perform consistently better with Gorder than with random ordering. Clearly, ordering strongly affects the performance of BFS-like algorithms. In Section 4, we propose a novel ordering scheme that benefits BFS-like algorithms.

3 BFS-Aware Static Cache

In Section 2.3, we discussed how the page cache is ineffective and that simply growing its size does not help BFS-like algorithms. In this section, we propose a different caching scheme to help improve the performance of BFS-like algorithms.

Aside from the typical cache that a graph engine or the system software manages, we propose to have a separate static cache, which we call the BFS-Aware Static Cache or BASC. BASC is loaded with the edge lists of some pre-selected vertices before the algorithm starts. Hence, there is overhead involved with the initial selection process, which we describe later in this section. Also, unlike a typical page cache that dynamically stores and replaces edge lists as they are accessed, BASC is static, that is, the cache contents do not change throughout its execution and no replacement is involved.

As the edge list of only some selected vertices are statically stored in BASC, it is important to identify the vertices that are likely to improve performance so that they can be stored in BASC. Note that naively storing the edge lists of frequently accessed vertices does not suffice. This is because, as was discussed in Section 2.2 and shown with Figure 1, the number of accesses to the edge lists of each vertex is similar for all vertices. Thus, our approach is to consider the interaction between BASC and the page cache, which we elaborate below.

The key optimization point with BASC is memory utilization. While BASC is separate cache space, we do not make use of extra space, but rather take space from the page cache, that is, reduce the page cache size, and use this space for BASC. Thus, when selecting the vertices for BASC, our goal is to utilize the space for BASC much more efficiently than when used as a page cache. For example, consider a case in Figure 4, where u's neighbor vertices (i.e., their edge lists) are about to be retrieved. (Note that in real graphs, an edge list can be composed of in-bound and/or out-bound edges. Here, we show an illustration using out-bound edges.) Notice that while the edge lists of u's neighbor vertices x, y, and z are stored in page A, the edge list of vertex v is stored in page B. Thus, while a cached page A would be well used, page B, on the other, would be retrieved only to access the edge list of v with the rest of the page being brought in for naught. In such a case, v would be pre-selected and it's edge list stored in BASC, so that the entire page containing the data (page B, in this case) need not be retrieved to the page cache during execution. Selecting vertices for BASC in this manner to improve overall memory utilization is formally discussed in the next section.

3.1 The Vertex Pre-Selection Problem

We formulate the problem of pre-selecting vertices for BASC as a problem of minimizing the overall weighted I/O requests for accessing edge lists. To define the problem, we first make the following assumptions:

Assumption 1 The neighbor vertices of each vertex are accessed simultaneously. Thus, their edge lists are retrieved at the same time.

Assumption 2 The number of edge list requests for each vertex is equivalent among all the vertices.

Assumption 3 Each edge (u,v) probabilistically issues a request to access the edge list of target vertex v. Due to Assumption 2, the probability of issuing the request is inversely proportional to v's in-degree.

We now define the problem of selecting vertices for BASC as a problem of minimizing the overall weighted requests:

$$\underset{C}{\text{minimize}}\ F(C)=\sum_{v\in V}\sum_{\substack{(v,u_1)\in E\\u_1\notin C}}\left(\frac{1}{r(u_1)d_i(u_1)}\frac{1}{\sum_{\substack{(v,u_2)\in E\\u_2\in P(u_1)\\u_2\notin C}} d_o(u_2)}\right)$$

$$\text{subject to } \sum_{v\in C} deg(v)\le M,\ \sum_{v\in C} deg(v)\ge M-\epsilon, \text{ where}$$

- ϵ is a small positive number
- C represents the set of cached vertices
- E is the set of all edges in the given graph
- $P(u)$ is the set of vertices whose edge lists are stored in the same page as vertex u
- $r(u)$ is the expected number of requests to the page where u is stored, which we assume to be proportional to the number of vertices whose edge lists are stored in the page
- $d_i(u)$, $d_o(u)$ are the in- and out-degree of vertex u, respectively
- M represents the size of available memory for BASC.

Function F represents the penalty accrued by misuse of pages in the cache. If the cache is fully utilized, there is no penalty. F increases as the cache is more and more underutilized. Hence, our goal is to minimized F.

More concretely, F iterates over all the edges in a graph; for each vertex, we take every edge $((v,u_1)\in E)$ that is not in the cache $(u_1\notin C)$ and adds up the penalty, which is represented by the terms in parenthesis. Within the left term in the parenthesis, $1/d_i(u_1)$ represents the probability of issuing the request for a given edge. If the page is already in the page cache, we do not need to issue the request (realized by $u_1\notin C$). To consider only those pages not in the cache, we include the other term $1/r(u_1)$, as we assume that the probability of the page holding u_1 to be in the cache is proportional to the number of expected requests to the page. Thus, the left term in the parenthesis represents the probability of issuing a request over the edge (v,u_1) for u_1 not in the cache. The right term in the parenthesis represents the actual penalty incurred, which is inversely proportional to the utilization of the page. Page utilization is computed as the summation of $d_o(u_2)$ because under Assumption 1, the neighbor vertices of each vertex are accessed simultaneously. The right term, $(1/\sum d_o(u_2))$, is minimized for (v,u_1) if all the other neighbor vertices (u_2) of v is stored in the same page as u_1 and they tightly fit in a single page.

Our intent here is to minimize the overall *weighted*, that is, penalized, requests. If we were to simply minimize the number of requests, we just need to cache the vertices in descending order of their degrees. In the evaluation, we demonstrate that our approach results in better performance than simply caching the vertices in degree order.

The above optimization problem is an integer programming problem. As the objective function $F(C)$ is nonlinear and non-convex, the problem is NP-hard [13]. Thus, a fast algorithm that provides an optimal solution does not exist. We propose a heuristic algorithm to solve this problem in the next section.

Algorithm 1: Greedy Vertex Selection (GVS) for BASC

```
   Input: G = (V,E), M: BASC size, K: iteration number
   Output: A set of selected vertices C
1  Function SelectVertices (G=(V,E), K, M )
2      C = ∅,   m = 0
3      for k := 1 to K do
4          for v ∈ V do
5              T[v] = 0
6          for v ∈ V do
7              for u1 ∈ neighbor(v) \ C do
8                  t = 0
9                  for u2 ∈ neighbor(v) ∩ P(u1) \C do
10                     t ← t + deg_o(u2)
11                 T[u1] ← T[u1] + 1/r(u1) · 1/t · 1/deg_i(u1)
12         while T ≠ ∅ do
13             u ← n ∈ T with minimum T[n]/deg_o(n)
14             if m + deg_o(u) ≥ (k/K)M then
15                 break;
16             C ← C ∪ {u},   m ← m + deg_o(u)
17     return C
```

3.2 Vertex Selection for BASC

We now present a heuristic algorithm for selecting vertices for BASC. Our algorithm, called Greedy Vertex Selection (GVS), takes a greedy approach based on the profits per cost for the vertex. In particular, a vertex is selected to be cached if the overhead, that is, the penalty, of the request for a vertex is high, where the penalty calculations are based on $F(C)$ described in Section 3.1.

Algorithm 1 shows the overall procedure of GVS. It takes as input G, the input graph, M, the memory size available for BASC, and K, the number of iterations, and

at each iteration selects vertices for M/K amount of memory. The outer-most Σ in $F(C)$ is covered by lines 6–11 and the next Σ by the `for` loop in line 7. The `for` loop in line 9 represents the Σ in the denominator of the second term within the parenthesis. Finally, line 11 represents the calculations for the two terms in the parenthesis for $F(C)$. Instead of summing up the penalties, GVS attributes the computed penalty to target vertices of individual edges – $T[u_1]$ in line 11. After each iteration, the penalty of each vertex is normalized by its degree (line 13). GVS selects the vertices with the highest normalized penalty whose degrees amount to $1/K$ of Basc and puts them in Basc. This is repeated for K iterations to completely fill Basc.

As K becomes larger, we select smaller number of vertices at each iteration and more frequently compute the changes in I/O penalty as the result of the selection. Thus, larger K gives more fine-grained and accurate vertex selection, but incurs more computational overhead.

The time complexity of GVS is $O(K(|E|+|V|))$ as the computation of the page utilizations in lines 10–11 can be calculated once and re-used per each vertex v, and the loop in lines 12–16 can be implemented using selection algorithms for finding the k'th smallest number.

We can further optimize the algorithm by computing the page utilization only for those vertices that are affected by the selection in the previous iteration. The vertices that require re-computation are those that are stored in the same pages as the selected vertices and are in neighbor relations. Let us now consider the complexity of GVS with this optimization. The number of selected vertices in an iteration is $O(M/K)$, if the selected ones have the same degree. The number of vertices for re-computation is proportional to the number of disk pages that the selected vertices are stored in. In one extreme, all the selected vertices may be in a same disk page, while in the other extreme, all may be in separate pages. If the layout of the graph is carefully ordered to improve locality, the selected vertices in an iteration will tend to be grouped and stored in a small number of pages. Thus, we derive the complexity assuming that the number of pages is bounded by the square root of the selected vertices, which is in between the two extremes. Then, the cost for the re-computation in lines 6 through 11 is $O(\sqrt{M/K})$. Furthermore, the sorting and selection of the M/K vertices in lines 12 through 16 can be done incrementally using a heap data structure, thus its complexity is $O(M/K \cdot log(|V|))$. Thus, the complexity of GVS is $O(|E|+|V|+\sqrt{K \cdot M}+M \cdot log(|V|))$, where the first two terms are for the first iteration and the rest are for the remaining $K-1$ iterations. In Section 5, we experimentally show that our complexity analysis for GVS is reasonable and that GVS time is roughly proportional to $\sqrt{K}$ and M.

4 Bringing New Order

In Section 2.4, we showed how ordering affected the performance of BFS-like algorithms and presented the need for effective graph layouts. In this section, we present an ordering scheme that we call Neighborhood Ordering (Norder, for short) that is tailored to BFS-like algorithms. Before so doing, we first present the I/O cost model that forms the basis for the development of Norder.

4.1 Modeling I/O Cost

In all BFS-like algorithms, the vertices that are activated in a particular iteration are the neighbor vertices of the frontiers of the previous iteration. How these activated vertices are ordered on disk substantially affects I/O performance. If these vertices could be stored together, the number of I/O requests could be reduced, leading to improved performance.

From this intuition, we empirically derive the following cost model for BFS-like algorithms, where cost is the edge list access cost, which we want to minimize:

$$Cost = \sum_{v \in V} deg(v) \cdot \sigma^2(nbr(v)) \qquad (1)$$

where $deg(v)$ is the in-degree of vertex v and $\sigma^2(nbr(v))$ is the variance of v's neighbor vertex IDs, assuming CSR format. The first term, $deg(v)$, which implies that cost increases with higher in-degree, that is, with more neighbors, comes from empirical and algorithmic analysis. Consider a vertex with high in-degree. Such a vertex is likely to be accessed in an early iteration of BFS traversal. Thus, it is also likely that the majority of its neighbor vertices have not yet been traversed, which, in turn, incurs access to new edge lists, and thus, increases I/O cost. In contrast, a vertex with low in-degree is likely to be traversed in a later iteration. At this point of traversal, it is also likely that the majority of its neighbors would have already been traversed and thus, not incur any more I/O cost. Thus, we conclude that I/O cost is proportional to the degree of the vertex. The second term, $\sigma^2(nbr(v))$ is the overhead of I/O based on the neighbors' vertex ID variance. That is, neighbors whose vertex IDs show large variance are likely to be scattered across the disk, in contrast to those whose IDs are close together. Neighbors widely scattered along the disk will naturally incur more I/Os to have them retrieved.

To assess the model's accuracy, we compare the costs estimated by the model and the actual execution times for three BFS-like algorithms with the datasets YT, FL, and LJ (which we describe in detail in Section 5). For each algorithm and dataset, we generate 20 graph orderings that yield different I/O costs. This is done by, first, applying three orderings – PageRank-sorted, Gorder, and the original ordering, and then, incrementally and partially shuffling the vertex IDs of each ordering.

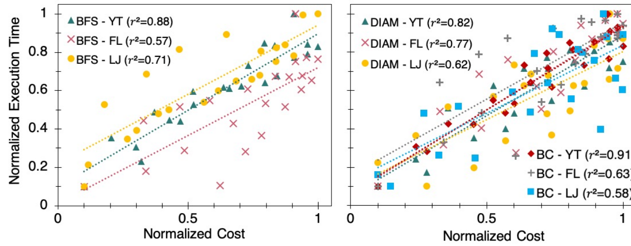


Figure 5: Regression of I/O cost model for three BFS-like algorithms with three datasets YT, FL, and LJ.

Figure 5 shows the results, where each data point represents one of the 20 graph orderings with its position determined by the cost of the model (x-axis) and the execution time (y-axis), both normalized to the maximum value on each axis. The lines are the results of applying linear regression on the data points, with the r^2 values of each dataset shown in parentheses. We see that our cost model results in reasonably high r^2 values, meaning that the estimation is quite accurate.

4.2 Neighborhood Ordering

Based on the cost model in the previous section, we propose a simple, yet effective graph ordering that we call *Neighborhood Ordering* or Norder. The key component of Norder is to simply assign consecutive IDs to the neighbors of all vertices, in particular, to the neighbors of high in-degree vertices. This simple strategy has the effect of decreasing the variance of the neighbor vertex IDs of high in-degree vertices resulting in increased locality, thus minimizing the overall cost of Equation 1.

To reorder a graph with Norder, we first arrange the vertices in descending order of their in-degrees. Then, starting from the vertex with the highest in-degree going downward, we perform a bounded breadth first search and assign a vertex ID in the traversed order. The depth bound for the traversal is set to two as we empirically found it to be effective for overall performance. Depth beyond two showed minimal performance gains, but incurred high overhead for ordering.

This simple ordering scheme is inexpensive to compute compared to other schemes such as Gorder, yet effective for disk-based graph engines. For example, it takes less than five minutes to compute Norder for the Twitter graph with 1.9 billion edges, while Gorder takes more than three hours. We quantify this and other performance issues in the next section.

5 Evaluation

We evaluate the effect of Basc and Norder with six BFS-like algorithms – breadth first search (BFS), measuring diameter of graph (DIAM), betweenness centrality (BC), shortest paths (SP), all-pair shortest paths (APSP), and weakly connected components (WCC). These algorithms represent, to the best of our knowledge, all the BFS-like algorithms in the field, except for Influence Maximization (IM). IM, however, is simply a repetitive execution of BFS, hence omitted from our evaluation [16].

BFS and SP are written as described in Pregel [25]. DIAM runs BFS multiple times, first from a random vertex and then from the vertices with the maximum distances in previous runs. For APSP, we sample 128 source vertices and compute the distances from those sources using a modified SP that computes the distances from a group of source vertices, the same as it is written in Graphene [21]. BC implements the algorithm proposed by Brandes [4]. It runs SP from each source vertex and counts the number of paths passed for each vertex. This is repeated for all the source vertices. As computation is intense, an approximate approach is taken by computing the centrality scores with 128 randomly sampled source vertices [5]. WCC is implemented in the typical manner of propagating component IDs for each vertex and then computing the minimum IDs.

Table 1: Datasets used in evaluation.

Graph	V	E	Graph	V	E
Youtube (YT)	3.2M	9.4M	LiveJournal (LJ)	4.8M	68M
Flickr (FL)	2.3M	33M	Twitter (TW)	53M	1.9B
Wikipedia (WK)	18M	172M			

The experiments are conducted with five real-world networks that are publicly available from KONECT [36] and are shown in Table 1. The datasets include one large network, Twitter (TW) and two relatively small graphs, Youtube (YT) and Flickr (FL), that are used to understand the scalability of our techniques.

We implement and evaluate our optimizations mainly on FlashGraph, a semi-external graph engine optimized for SSDs. We choose FlashGraph because 1) it is a representative semi-external graph engine, 2) it is recently developed thus, most known I/O optimizations are provided, and 3) it is actively maintained and core graph algorithms are already implemented in the system. Of the six algorithms, SP and APSP are our own implementations that we added, while the other four are those provided by FlashGraph. Additionally, we test our optimizations on Graphene [21], an SSD-based graph engine optimized with fine-grained I/O management. Details of these experiments are discussed in Section 5.4.

For all experiments, we run the algorithms with eight computation threads and a separate single I/O thread. We run the experiments ten times and report the average execution times as well as the standard deviation. All the experiments are performed on a machine with Intel Xeon E5-2683 v4 having 128GB physical memory running Ubuntu 16.04. Of the 128GB memory the system is configured to use only a portion of the memory (maximum 10GB) for caching. The actual cache size

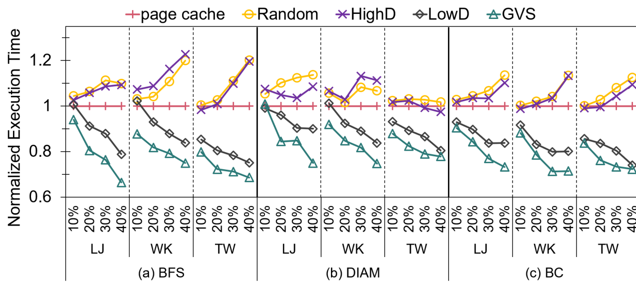


Figure 6: Execution times of graph algorithms on SSD with Basc using different vertex selection schemes normalized to the page cache only case. We vary the cache size to be 10% to 40% of the input graph size.

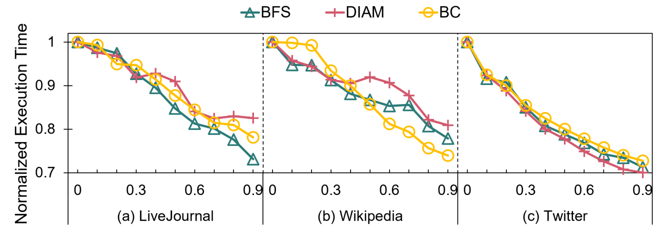


Figure 7: Execution times with varying Basc and page cache size ratios normalized to the page cache only case. The x-axis represents the Basc usage portion.

used varies for the algorithm and dataset, hence, for each experiment, we specify the cache size relative to the input graph size. Note that our goal is to show the effectiveness of our techniques on disk-based graph engines. Hence, we intentionally control memory to exercise the disks. The memory used is controlled by fixing the memory size that each part of the graph system uses and monitoring the total amount of memory used.

An Intel 400GB SSD connected via the SATA 6.0Gb/s interface is used as external storage as this is a common setting. We also run the same experiments on a typical HDD setting, but do not present the results in the interest of space and as the results do not deviate much from the results for the SSD. For each experiment, only a single disk is used. Unless otherwise stated, we set the page size to be 8KB, which is the typical access unit for NAND flash-based SSDs [8]. We use the page cache implemented in the graph system and hence, no OS kernel modification is involved. We completely clear the page cache for each experiment, and the OS page cache is turned off by setting the *O_DIRECT* flag for I/O operations so as to remove the system effect.

Note that, unless otherwise mentioned, we present results only for three datasets, LJ, WK, and TW for the BFS, DIAM, and BC algorithms only. This is done in the interest of space and as the results for the other datasets and algorithms show similar trend. However, we present statistical numbers for all algorithms and datasets and, where there are unique points of interest, we explicitly elaborate on those points separately.

5.1 Evaluation of Basc

In this section, we evaluate Basc and the Greedy Vertex Selection (GVS) algorithm in various settings. In evaluating Basc, we compare the 'page cache plus Basc' setting with the 'page cache only' setting, where both settings use the same amount of memory. To compare with GVS, we test three other selection schemes for Basc: selecting vertices with high degrees (HighD), selecting vertices with low degrees (LowD), and selecting vertices randomly (Rand), of which the last is used as the baseline. For GVS, we set the total number of iterations (K in Algorithm 1) to be 1,000 for YT, FL, LJ, WK, and 200 for TW. (Even though we do not show results for YT and FL, the settings for all experiments are presented for reproducibility.) As was previously discussed, with large K values, a smaller number of vertices are selected per iteration. This results in more frequent computation of I/O penalty changes, thus increased computation overhead. Hence, for feasibility reasons we choose a smaller K value for the large input graph TW. For the experiments in this section we do not apply any reordering algorithm and use the input graphs as they are given in KONECT [36]. We evaluate the combined effect of ordering and Basc separately in Section 5.3.

Evaluation Result. Figure 6 shows the results for experiments where the page cache size is set to 5% of the graph size and an additional 10% to 40% of the graph size is provided for Basc. As mentioned earlier, for the 'page cache only' setting (denoted 'page cache'), the page cache size is equally set, that is, 15% to 45% of the graph size is used. All results are shown normalized to that of 'page cache only'. The results presented here exclude the time to load Basc, which must be done before the algorithms are executed. We discuss this and other forms of overhead in the last part of this section.

Overall, Basc with GVS shows substantially better performance than the other schemes. We see that low-degree selection generally is a sound choice performing, in most cases, better than page cache, though, we do observe cases where it does worse. Overall, for all algorithms and datasets (including those not shown here), Basc with GVS is the clear winner, being 28.87% faster on harmonic average than page cache for all the cases.

To compare the efficiency of 'page cache' and Basc with GVS, we vary the ratio of the page cache and Basc sizes and measure their performance. Figure 7 shows the results as the total size of the two caches is set to 20% of the graph size and as the size of Basc varies from 0% to 90% of the total cache size. We see that the performance consistently improves as Basc size is increased. This is because BFS-like algorithms show

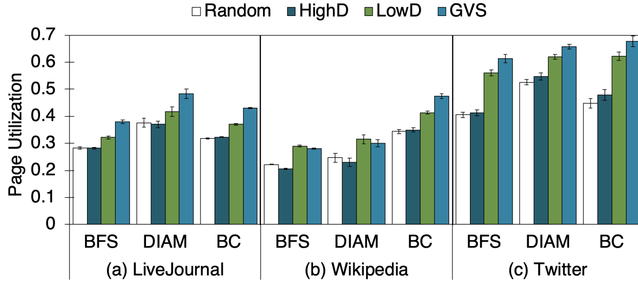


Figure 8: Mean and standard deviation of page cache utilization when executed with BASC with Random, HighD, LowD, and GVS algorithms.

poor temporal locality and using the space as BASC is more effective than using the space as a page cache.

Effect on page cache utilization. To verify that BASC with GVS improves page cache utilization, we evaluate BFS-like algorithms with BASC and measure the page cache utilizations for the different vertex selection algorithms (GVS, HighD, LowD, and Rand).

Page cache utilization is measured as follows. For the entire execution of the algorithms, for each page retrieved to the page cache, individual 64 byte granularity units of the page are monitored throughout while the page resides in the page cache. Utilization of the page, calculated when the page is evicted, is the fraction of the total accessed units within the page size. Page cache utilization that is reported is the average of all the pages evicted as well as those still residing in the page cache at the end of the algorithm execution. Note that the contents in BASC are not considered in these calculations. Essentially, this value tells us how efficiently contents brought in to the page cache are being used.

Figure 8 compares the page cache utilizations for different vertex selection algorithms when the page cache and BASC size is 5% and 20%, respectively, of the input graph size. For all the graphs across the three algorithms, GVS shows highest page cache utilization for most cases. For all algorithms and datasets experimented with (for the same page cache and BASC sizes as above), GVS shows 33.8% higher utilization than random selection in harmonic mean, and in the best case, is almost twice that of random selection (FL with BFS, not shown), with the worst case, being equivalent (YT with DIAM, also not shown). Compared to LowD, GVS shows up to 22.5% higher utilization (FL with APSP). However, there are also occasions where low-degree selection shows slightly higher utilization (WK with BFS and DIAM).

Vertex Selection and Loading Overhead. There are two sources of overhead in deploying BASC. One is the cost for running GVS to select the vertices to load and the other is the overhead to actually load the selected vertices into BASC. Both must be executed before the algorithm is executed. We quantify these overheads by measuring the running time of Algorithm 1 separately with varying K from 1 to 1000, as well as the time to load to BASC. Table 2 shows the results when BASC size is 20% of the dataset. We observe that vertex loading overhead is generally lower compared to the vertex selection overhead. For vertex selection, as we increase the value of K, the execution time increases sublinearly. The results also show that the selection time increases proportionally to the graph size, or more accurately, BASC size. (Refer to Table 1 for the characteristics of the dataset and note that we are setting BASC size (M) to be 20% of input graph size, which is proportional to the number of edges.)

Table 2: Execution times of running GVS and loading the selected vertices to BASC (unit: seconds).

	K	YT	FL	LJ	WK	TW
GVS	1	4.6	4.9	7.8	19.8	132
	10	5.9	7.6	11.3	29.2	321
	100	16.7	24.7	26.3	76.4	1581
	1000	94.8	103	146	449	5612
Loading		2.5	3.2	4.3	6.2	44.8

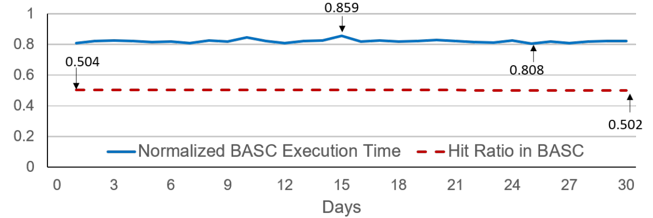
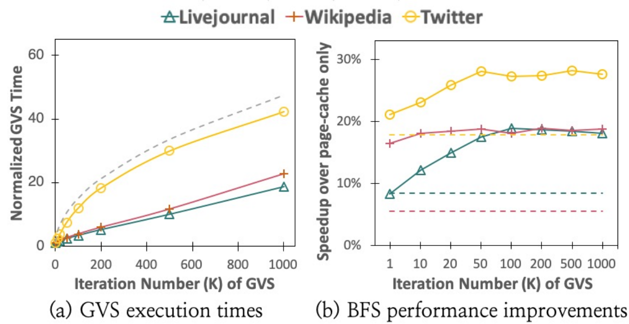


Figure 9: Efficiency of BASC over a graph (LJ) that changes 0.2% each day. Execution time of BFS with BASC is normalized to that of page cache only each day.

Although the selection and loading times are not negligible, once loaded, the algorithm of choice is executed 100s, if not 1000s or even more times [11,42], more than compensating for the overhead for selection and loading. For example, running of DIAM a hundred times on LJ or BC just ten times on TW with BASC improves the overall running time even with the selection and loading overhead. Furthermore, this selection process can be run in the background independent of and without influencing the execution of the graph algorithms.

More importantly, these actions need to run only sparingly. Reports have shown that the social graph in Facebook changes by 0.2% a day [9], which means that in a one month period the graph would only differ by 6%. To quantify how much effect small changes in the input graph have on BASC, we conduct a series of experiments where we change the graph (LJ) by 0.2% over 30 times to simulate changes in a one month period [9].



(a) GVS execution times (b) BFS performance improvements

Figure 10: The sensitivity of GVS on iteration number K. (a) Execution time normalized by that with $K = 1$, where the dotted gray line represents $1.5 \times \sqrt{K}$. (b) Speedup over page cache only where the solid lines are for GVS and the dotted lines are for LowD.

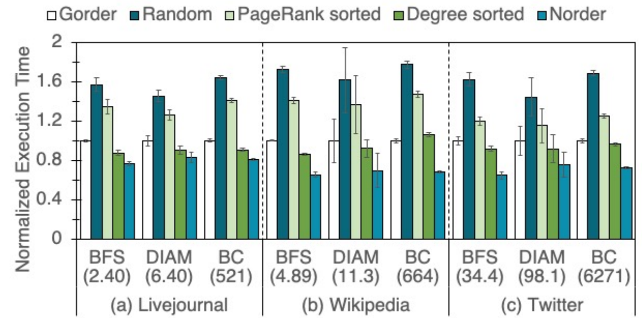


Figure 11: Mean and standard deviation of execution times of five graph orderings normalized to that of Gorder. The numbers in parenthesis below each algorithm are the absolute execution time, in seconds, of the reference, in this case, Gorder. (Note that we use similar presentation format in subsequent figures.)

We randomly add or remove edges with preferential attachment [3] where the ratio to add and remove is 8 to 2. Then, we run BFS with Basc, set to 20% of the data size, for which the selection and loading is done for the initial graph. Figure 9 shows the results of the experiments where the execution times with Basc is normalized to those of page cache only for each day. We observe that the relative performance of Basc over page cache only is nearly constant with only a 5% difference – maximum 19.2% and minimum 14.1%. In addition, the hit ratio of Basc, which is measured as the number of edge list access in Basc over the total number of edge list requests, declines slowly each day from 50.4% to 50.2%. This tells us that selection and loading can be performed over long periods, for example, once every month incurring only minimal overhead.

Sensitivity of GVS on iteration number K. As GVS execution time and its selection outcome rely on the iteration number K, we evaluate their sensitivity on K. We run GVS with varying values of K, starting from 1 up to 1000, and measure the GVS execution time. The size of Basc is set to be 20% of the input graph size. Figure 10(a) plots the results normalized to the execution times with $K = 1$. We observe that execution time increases by less than 45×, even for $K = 1000$, and that the plots for the three input graphs are all below the $1.5 \times \sqrt{K}$ line represented by the gray dotted line.

We also measure the performance of Basc with GVS as K is varied. Figure 10(b) shows the performance improvements over the page cache only settings. The solid lines are for Basc with GVS and the dotted lines are for Basc with LowD (low-degree vertex selection). We observe that Basc with GVS performs consistently better than LowD even when $K = 1$. Moreover, as K increases, the performance improvement by GVS quickly saturates, that is, the performance improvement with $K = 50$ is almost the same as that with $K = 1000$. Hence, for extremely large graphs whose GVS overhead can be potentially quite high, we can reduce the value of K to trade off the performance improvements of running the graph algorithms and GVS running time.

5.2 Evaluation of Neighborhood Ordering

We now consider the performance impact of Norder. For comparison, we use Gorder, the state-of-the-art graph ordering scheme for in-memory graph analysis. Also, we evaluate three other ordering schemes: PageRank sorted ordering, degree sorted ordering, and random ordering. For Gorder we set its parameter w (window size) to be the average number of edge lists in a single page for all the algorithms. Note that only the page cache is used and Basc is not deployed in these experiments. The size of the page cache is set to 25% of the input graph size.

Figure 11 shows the performance of the algorithms on an SSD normalized to Gorder, with the absolute execution times (in seconds) of Gorder also presented in the parentheses below each algorithm name for reference. We use this format of presentation for subsequent results as well. For all three algorithms, graph ordering has strong influence, with Norder performing the best. For all algorithms and datasets, Norder is 31.3% and 68.5% faster in harmonic mean than Gorder and PageRank sorted ordering, respectively. For all algorithms and datasets Norder showed fastest performance except for WCC with TW and FL, for which Degree sorted order was fastest. This is due to the characteristics of WCC. In the later iterations of the algorithm, a small number of vertices linger on; these vertices typically have small in-degrees, hence storing them closely on disk improves the performance of the page cache for WCC.

Cost of ordering: The cost of applying Norder is much lower than that of Gorder. Table 3 compares the computation times of the two ordering schemes. As Gorder stores the input graph in main memory to compute the ordering, all data is loaded to the 128GB memory in our

Table 3: Computation times (unit: seconds)

	YT	FL	LJ	WK	TW
Gorder	12.5	39.6	45.6	169.3	11687.1
Norder	2.0	2.7	7.2	16.9	243.5

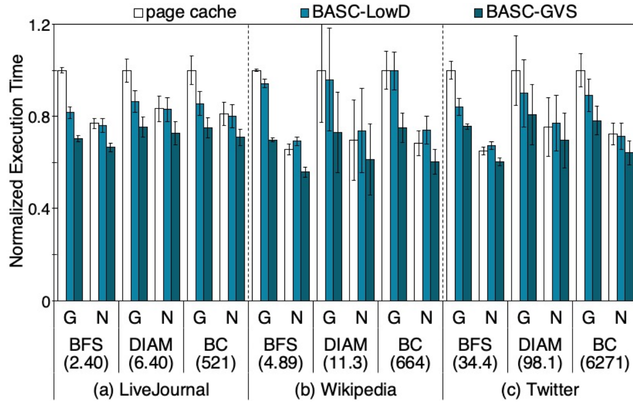


Figure 12: Mean and standard deviation of execution times of three caching schemes with Gorder and Norder normalized to that of page cache with Gorder.

system for these measurements. All orderings are computed using a single thread as this is how it is provided with the open-sourced Gorder. We observe that Norder is 6 to 14 times faster than Gorder for small graphs, while for the large graph TW, it is close to fifty times faster. If we consider the ordering time with the execution time of the algorithms, we can see that BFS-like algorithms benefit even more with Norder.

5.3 Combining Basc and Norder

Now we evaluate the overall performance gain by applying the two optimizations together. For comparison we also perform experiments with Gorder and two caching schemes, page cache and Basc with LowD. For caching, the default setting of 5% of the input graph size is used for the page cache and an additional 20% is added on as Basc or the page cache.

Figure 12 shows the performance results for all combinations of ordering and caching. The two optimizations together noticeably improve the performance of all the algorithms. Overall, for all algorithms and datasets (including those not shown here) Basc with Norder is 1.54 times faster than page cache with Gorder in harmonic mean. In the best case, Basc with Norder is 2.56 times faster for APSP with YT and in the worst case, it is 1.37 times faster for DIAM with LJ.

More importantly, however, the results demonstrate that the two optimizations can be synergistic. For example, low-degree vertex selection sometimes brings about performance degradation compared to page cache (WK with BFS and DIAM), implying that the two optimizations (LowD and Norder) do not interact well. However, GVS consistently and substantially improves performance in all cases with the two orderings, especially with Norder.

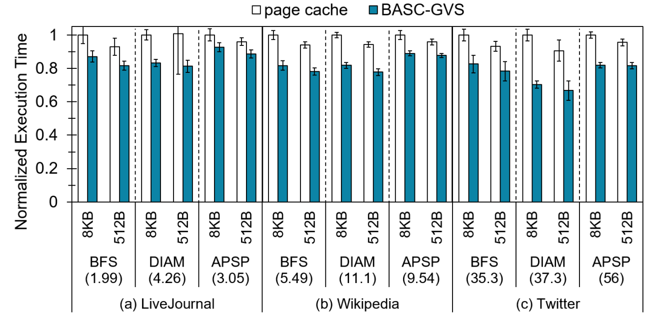


Figure 13: Mean and standard deviation of execution times of BFS-like algorithms in Graphene, with and without Basc, normalized to that of page cache only case with 8KB page size.

5.4 BASC with Graphene

The problem of low page cache utilization for disk-based graph systems was studied by Liu and Huang [21]. In their proposed graph system, Graphene, they address this problem by supporting finer-grained I/O. Specifically, Graphene stores input graphs in 512-byte pages instead of 8KB and applies bitmap-based request management to reorder and merge I/Os.

In this section, we incorporate Basc on Graphene and observe its effect. To do so, we simply modify the page cache mechanism within Graphene to accommodate Basc. Here we evaluate the effect of Basc with three algorithms – BFS, APSP, and DIAM. The first two are provided in Graphene and we implement DIAM for our experiments. We were unable to implement BC due to the complexity of Graphene's interface. Note that in Graphene, APSP is implemented to run BFS from 32 random sources and stores the result in a 4-byte attribute, of which each bit indicates if the traversal from the corresponding source vertex is reached. We compare performance with and without Basc in Graphene with the typical 8KB and the 512-byte page sizes. The page cache is set to 30% of the input graph without Basc and with Basc, the page cache is set to 10%, plus 20% space set for Basc, along with the default thread setting. Norder is not considered in these experiments as this is an optimization independent of Graphene.

Figure 13 shows the evaluation results. The results are normalized to the 8KB 'page cache only' results for every algorithm for each dataset. For most of the results the average performance of fine-grained management is better than the coarse-grained 8K page size, though for some, the variance for fine-grained management is larger. We observe that overall, Basc provides similar

improvements with Graphene showing that BASC is orthogonal to Graphene's fine-grained I/O optimizations.

6 Related Work

Disk-Based Graph Engines. GraphChi is the first disk-based graph engine [19]. Its *Parallel Sliding Windows* helps it run efficiently on HDDs. TurboGraph is a disk-based graph engine for SSDs [12]. Its *pin-and-slide* technique overlaps random I/O with CPU computation. TurboGraph and other graph systems such as GTS and GraphZ [17, 41] that use the page cache for random I/O can take advantage of BASC or Norder.

While the vertex-centric computation model is widely used, an alternative *edge-centric* computation model was recently proposed for disk-based graph systems; this model sequentially streams edges into memory to eliminate random disk access [24, 32, 43]. While the edge-centric model shows good performance for algorithms accessing the entire graph repeatedly, its performance for BFS-like algorithms is not as efficient.

In semi-external graph engines, vertex attributes are stored in main memory for fast updates [18, 30, 34, 40]. Pearce et al. proposed asynchronous optimization techniques for graph traversal algorithms for semi-external graph processing [30]. FlashGraph implements several I/O optimizations for SSDs and SSD arrays such as merging I/O requests and overlapping I/O and computation [40]. Building on top of these optimizations, our methods improve BFS-like algorithms having poor I/O locality even with those previous optimizations.

Several other I/O optimizations have recently been proposed. Vora et al. employs a dynamic partitioning scheme that prevents loading unnecessary edges [37]. GridGraph supports 2D edge partitioning [43]. In Graphene, a bitmap based I/O optimization is applied to merge small I/O requests [21]. Our proposed optimization techniques are applicable on top of these I/O optimizations as we have shown with Graphene.

Main Memory Graph Processing. For large-scale graph processing, the vertex-centric computation model was first proposed in Pregel [25], a distributed in-memory graph system. GraphLab and its successor PowerGraph is a distributed machine learning and graph analysis system with the vertex-centric model [11, 22]. SociaLite is a Datalog-based query language for distributed graph analysis [33]. Green-Marl is a domain-specific language for writing parallel graph algorithms for shared-memory [14]. Galois supports an implicitly parallel vertex iterator for graph processing [29].

Wei et al. studied graph ordering for main memory graph processing [38]. They proposed Gorder that optimizes the locality of accessing vertex attributes. While Gorder is designed for main memory systems, Norder is for disk-based graph engines. Norder is based on an I/O cost model and its optimization that we derive for BFS-like algorithms on disk-based graph engines.

7 Conclusion

In this paper, we conducted an analysis of BFS-like algorithms running on disk-based graph systems. We showed that BFS-like algorithms have poor I/O performance and the page cache in existing systems is not effective. To supplement the page cache, we proposed a BFS-Aware Static Cache or BASC that stores edge lists of a select set of vertices in memory aside from the page cache. We formulate the problem of selecting the optimal set of such vertices as a problem of maximizing overall I/O efficiency of BFS-like algorithms. As this problem is NP-hard, an approximate algorithm, called Greedy Vertex Selection (GVS), is developed. Also, based on our analysis of BFS-like algorithms, we proposed an I/O cost model upon which we develop an efficient graph ordering scheme called *Neighborhood Ordering* (abbreviated Norder) that stores neighboring vertices closely on disk.

We implemented our methodologies in two well-known graph engines and evaluated them using five real-world graphs for six BFS-like algorithms. Through a vast set of experiments, we show that the execution of BFS-like algorithms can be improved with BASC and GVS compared to simply using the page cache. We also show that Norder is less costly to compute than Gorder, yet achieves considerable performance improvements over Gorder. Our experimental results show that the two optimizations collectively and synergistically provide substantial performance gains for BFS-like algorithms.

Acknowledgement

This work is supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B1016114), by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2016M3C4A7952635), by Basic Research Laboratory Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT Future Planning (MSIP) (No. 2017R1A4A1015498), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C2009476). The corresponding author is Jiwon Seo.

References

[1] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.

[2] Sam Ainsworth and Timothy M Jones. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing (SC 16)*, pages 1–11. ACM, 2016.

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Ulrik Brandes. A faster algorithm for Betweenness Centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[5] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys 15)*, pages 1–15, 2015.

[7] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C.S. Lui, and Cheng He. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE 15)*, pages 1131–1142, 2015.

[8] Michael Cornwell. Anatomy of a solid-state drive. *Communications of the ACM*, 55(12):59–63, 2012.

[9] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Walking in Facebook: A case study of unbiased sampling of OSNs. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM 10)*, pages 1–9, 2010.

[10] Debra S. Goldberg and Frederick P. Roth. Assessing experimentally derived interactions in a small world. *Proceedings of the National Academy of Sciences*, 100(8):4372–4376, 2003.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[12] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD 13)*, pages 77–85, 2013.

[13] Raymond Hemmecke, Matthias Köppe, Jon Lee, and Robert Weismantel. Nonlinear integer programming. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, pages 561–618. Springer-Verlag, 2010.

[14] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 12)*, pages 349–362, 2012.

[15] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[16] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD 03)*, pages 137–146, 2003.

[17] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 16)*, pages 447–461, 2016.

[18] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*, pages 830–841, 2016.

[19] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[20] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM 03)*, pages 556–559, 2003.

[21] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, 2017.

[22] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 2007.

[24] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the European Conference on Computer Systems (EuroSys 17)*, pages 527–543, 2017.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 10)*, pages 135–146, 2010.

[26] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Switching Theory, 1959*, pages 285–292, 1959.

[27] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. In *Proceedings of the International Workshop on Architecture for Graph Processing (AGP 17)*, 2017.

[28] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *International Journal of Multimedia & Ubiquitous Engineering*, 9(2):199–212, 2014.

[29] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 13)*, pages 456–471, 2013.

[30] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*, pages 1–11, 2010.

[31] Milo Polte, Jiri Simsa, and Garth Gibson. Comparing performance of solid state devices and mechanical disks. In *Proceedings of the Annual Workshop on Petascale Data Storage (PDSW 08)*, pages 1–7. IEEE, 2008.

[32] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 13)*, pages 472–488, 2013.

[33] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed SociaLite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

[34] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. Fog: A fast out-of-core graph processing framework. *International Journal of Parallel Programming*, pages 1–14, 2016.

[35] Alfonso Shimbel. Structural parameters of communication networks. *The bulletin of mathematical biophysics*, 15(4):501–507, 1953.

[36] The koblenz network collection. http://konect.uni-koblenz.de/.

[37] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of the USENIX Annual Technical Conference (ATC 16)*, pages 507–522, 2016.

[38] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 16)*, pages 1813–1828, 2016.

[39] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, 2016.

[40] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[41] Zhixuan Zhou and Henry Hoffmann. Graphz: Improving the performance of large-scale graph analytics on small-scale machines. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE 18)*, pages 1368–1371, 2018.

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, 2016.

[43] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC 15)*, pages 375–386, 2015.

From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

Sadjad Fouladi Francisco Romero Dan Iter Qian Li
Shuvo Chatterjee[+] Christos Kozyrakis Matei Zaharia Keith Winstein

Stanford University, [+]Unaffiliated

Abstract

We present gg, a framework and a set of command-line tools that helps people execute everyday applications—e.g., software compilation, unit tests, video encoding, or object recognition—using thousands of parallel threads on a cloud-functions service to achieve near-interactive completion times. In the future, instead of running these tasks on a laptop, or keeping a warm cluster running in the cloud, users might push a button that spawns 10,000 parallel cloud functions to execute a large job in a few seconds from start. gg is designed to make this practical and easy.

With gg, applications express a job as a composition of lightweight OS containers that are individually transient (lifetimes of 1–60 seconds) and functional (each container is hermetically sealed and deterministic). gg takes care of instantiating these containers on cloud functions, loading dependencies, minimizing data movement, moving data between containers, and dealing with failure and stragglers.

We ported several latency-sensitive applications to run on gg and evaluated its performance. In the best case, a distributed compiler built on gg outperformed a conventional tool (`icecc`) by 2–5×, without requiring a warm cluster running continuously. In the worst case, gg was within 20% of the hand-tuned performance of an existing tool for video encoding (ExCamera).

1 Introduction

Public cloud-computing services have steadily rented out their resources at finer and finer granularity. Sun's Grid utility (2005), Amazon's EC2 (2006), and Microsoft's Azure virtual machines (2012) began by renting virtual CPUs for a minimum interval of one hour, with boot-up times measured in minutes. Today, the major services will rent a virtual machine for a minimum of one minute and can typically provision and boot it within 45 seconds of a request.

Meanwhile, a new category of cloud-computing resources offers even finer granularity and lower latency: cloud functions, also called serverless computing. Amazon's Lambda service will rent a Linux container to run arbitrary x86-64 executables for a minimum of 100 milliseconds, with a startup time of less than a second and no charge when it is idle. Google, Microsoft, Alibaba, and IBM have similar offerings.

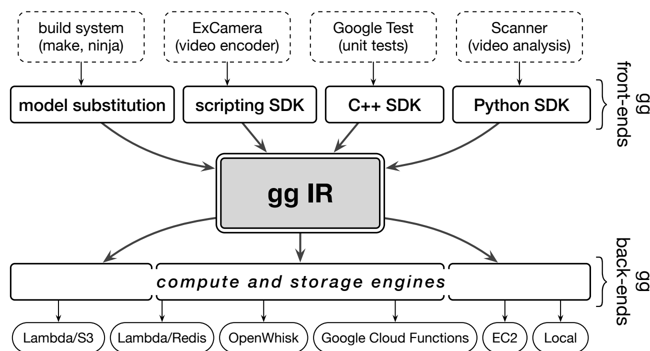


Figure 1: gg helps applications express their jobs as a composition of interdependent Linux containers, and provides back-end engines to execute the job on different cloud-computing platforms.

Cloud functions were intended for asynchronously invoked microservices, but their granularity and scale has allowed researchers to explore a different use: as a burstable supercomputer-on-demand. These new systems launch a *burst-parallel* swarm of thousands of cloud functions, all working on the same job. The goal is to provide results to an interactive user—much faster than can be accomplished on the user's own computer or by booting a cold cluster, and cheaper than maintaining a warm cluster for occasional tasks.

Recent work has validated this vision. ExCamera [15] and Sprocket [3] launch thousands of cloud functions, with inter-thread communication over TCP, to encode, search, and transform video files quickly. PyWren [23] exposes a Python API and uses AWS Lambda functions for linear algebra and machine learning. Serverless MapReduce [35] and Spark-on-Lambda [36] demonstrate a similar approach.

Unfortunately, building applications on swarms of cloud functions is difficult. Each application must overcome a number of challenges endemic to this environment: (1) workers are stateless and may need to download large amounts of code and data on startup, (2) workers have limited runtime before they are killed, (3) on-worker storage is limited, but much faster than off-worker storage, (4) the number of available cloud workers depends on the provider's overall load and can't be known precisely upfront, (5) worker failures occur

when running at large scale, (6) libraries and dependencies differ in a cloud function compared with a local machine, and (7) latency to the cloud makes roundtrips costly. Past applications have addressed only subsets of these challenges, in application-specific ways.

In this paper, we present gg, a general framework for building burst-parallel cloud-functions applications, by building them on an abstraction of transient, functional containers, or *thunks*. gg helps applications express their jobs in terms of interrelated thunks (hermetically sealed, short-lived containers that may reference the output of other thunks or produce other thunks as output), then schedules, instantiates, and executes those thunks on a cloud-functions service.

gg can containerize and execute existing programs, e.g., software compilation, unit tests, video encoding, or searching a movie with an object-recognition kernel. gg does this with thousands-way parallelism on short-lived cloud functions. In some cases, this yields considerable benefits in terms of performance. Depending on the frequency of the task (e.g., for compilation or unit tests every few minutes), cloud functions are also much less expensive than keeping a comparable cluster running continuously.

gg and other parallel execution systems. In its goals and approach, gg is kin with container-orchestration systems such as Kubernetes [5] and Docker Swarm [10], outsourcing tools like the Utility Coprocessor [12] and `icecc` [20], and cluster-computation tools such as Hadoop [38], Dryad [22], Spark [40], and CIEL [27].

But gg also differs from these systems in its focus on a new computing substrate (cloud functions), mode of execution (burst-parallel, latency-sensitive programs starting from zero), and target application domain (everyday "local" programs, e.g. software compilation, that depend on an environment captured from the user's own laptop).

For example, the "stateless" nature of cloud functions (they boot up with no dependable transient state) makes gg place a large focus on efficient containerization and dependency management: loading the minimal set of the right files into each container at boot-up. Cluster-computation systems like Dryad, Spark, and CIEL do not do this—although they can interface with existing code and systems (e.g., a video encoder or a database server), these components must be loaded in advance by the user on a long-lived compute node. Container systems like Kubernetes do this, but they are not aimed at efficient execution of a transient interactive task—gg is more than 45× faster than Google Kubernetes Engine at startup, and 13× faster than Spark-on-Lambda (Figure 7). We discuss related work more completely in Section 2.

1.1 Summary of Results

We ported four applications to express their jobs in gg's format: a description of each container, and how it depends on other containers, that we call the *intermediate representation*, or IR (§3). One of them does it automatically, by inferring the IR from an existing software build system (e.g., `make` or `ninja`). The rest write out the description explicitly: a unit-testing framework (Google Test [17]), parallel video encoding with inter-thread communication (ExCamera [15]), and object recognition using Scanner [30] and TensorFlow [1].

Compiling Inkscape

Tool	Time	Cost
single-core `make`	32m 34s	—
`icecc` to a warm 48-core EC2 machine	6m 51s	$2.30/hr
`icecc` to a warm 384-core EC2 cluster	6m 57s	$18.40/hr
gg to AWS Lambda	1m 27s	50¢/run

Figure 2: Compiling Inkscape using gg on AWS Lambda is almost 5× faster than outsourcing the job to a warm 384-core cluster, without the costs of maintaining a warm cluster for an occasional task.

We then implemented gg back-ends, which interpret the IR and execute the job, for five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, IBM Cloud Functions, and Google Cloud Functions) and three storage engines (S3, Google Cloud Storage, and Redis) (Figure 1).

For compiling large programs from a cold start, gg's functional approach and fine-grained dependency management yield significant performance benefits. Figure 2 shows a summary of the results for compiling an open-source software, Inkscape [21]. Running "cold" on AWS Lambda (with no pre-provisioned compute resources), gg was almost 5× faster than an existing system (`icecc`), running on a 48-core or 384-core cluster of warm VMs (i.e., not including time to provision and boot the VMs[1]).

In summary, gg is a practical tool that addresses the principal challenges faced by burst-parallel cloud-functions applications. It helps developers and users build applications that burst from zero to thousands of parallel threads to achieve low latency for everyday tasks. gg is open-source software and the source code is available at https://snr.stanford.edu/gg.

2 Related Work

gg has many antecedents—cluster-computation systems such as Hadoop [38], Spark [40], Dryad [22], and CIEL [27]; container orchestrators like Docker Swarm and Kubernetes; outsourcing tools like `distcc` [8], `icecc` [20], and UCop [12]; rule-based workflow systems like `make` [13], CMake [7], and Bazel [4]; and cloud-functions tools like ExCamera/mu [15], PyWren [23], and Spark-on-Lambda [36].

Compared with these, gg differs principally in its focus on targeting a new computing substrate (thousands of cloud functions, working to accelerate a latency-sensitive local-

[1] Current cloud-computing services typically take an additional 0.5–2 minutes to provision and boot such a cluster.

application task). We discuss how gg fits with the prior literature in several categories:

Process migration and outsourcing. The idea of accelerating a local application's interactive operations by using the resources of the cloud has a long pedigree; earlier work such as the Utility Coprocessor (UCop) also sought to "improve performance from the coffee-break timescale of minutes to the 15–20 second timescale of interactive performance" by outsourcing to a cloud VM [12]. gg shares the same goal.

gg's architectural differences from this work come from its different needs: instead of outsourcing applications transparently to a *single* warm cloud VM, gg orchestrates *thousands* of unreliable and stateless cloud functions from a cold start. Unlike UCop, gg is not transparent to the application—we require applications to be ported to express jobs in gg's format. In return, gg provides optimized orchestration of swarms of cloud functions and fault tolerance (failed functions are rerun with the same inputs). Unlike UCop's distributed caching filesystem, gg's IR, which is based on content-addressed immutable data, allows cloud workers to be provisioned with all necessary dependencies in a single roundtrip and to communicate intermediate values directly between each other.

Container orchestration. gg's IR resembles container and environment-description languages, including Docker [10] and Vagrant [34], and container-orchestration systems such as Docker Swarm and Kubernetes. In contrast to these systems, gg's thunks are designed to be efficiently instantiated within a cloud function, expressible in terms of other thunks to form a computation graph, and deterministic and defined by their code and data, allowing gg to provide fault tolerance and memoization. These systems were not designed for transient computations, and gg has much quicker startup. For example, starting 1,000 empty containers with gg takes about 4 seconds on a VM cluster or on AWS Lambda. Google Kubernetes Engine, given a warm cluster, takes more than 3 minutes (§5.1). Recent academic work has shown how to lower this overhead to provide faster cloud-functions services [28].

Workflow systems. Workflow systems like Dryad [22], Spark [40], and CIEL [27] let users execute a (possibly dynamic) DAG of tasks on a cluster. However, gg differs from these systems in some significant ways:

- gg is aimed at a different kind of application. For example, while Spark is often used for data analytics tasks, it is not commonly used for accelerating the sorts of "everyday" local applications that gg is designed for. No prior work has successfully accelerated something like "compiling Chromium" using Spark, and the challenges in accomplishing this (capturing the user's local environment and the information flow of the task, exporting the job and its dependencies efficiently to the cloud, running thousands of copies of the C++ compiler in a fault-tolerant way) are simply not what Spark does.
- gg uses OS abstractions: it encapsulates arbitrary code and dependency files in lightweight containers, somewhat similar to a tool like Docker. gg focuses on efficiently loading code and its minimal necessary dependencies on cloud functions that boot up with no dependable state. By contrast, systems like Dryad and Spark principally use language-level mechanisms. While their jobs can interface with existing software (e.g., the Dryad paper [22] describes how a node can talk to a local SQL Server process, and Spark jobs routinely invoke system binaries such as `ffmpeg`), these systems do not take care of deploying the existing code, worrying about how to move the container in a way that minimizes bytes moved across the network, etc. The user is responsible for loading the necessary code and dependencies beforehand on a pool of long-lived machines.
- gg is considerably lighter weight. In practice, attempts to port workflow systems to support execution on cloud functions (scaling from zero) have not performed well, partly because of these systems' overheads. Because of its focus on transient execution, gg carries an order-of-magnitude less overhead. For example, gg is 13× faster at invoking 1,000 "sleep 2" tasks than Spark-on-Lambda (Figure 7).
- gg supports dynamic data access (a function can produce another function that accesses arbitrary data) and non-DAG dataflows (e.g., loops and recursion). It does this while remaining agnostic to the application's programming language. For example, gg has no language-level API binding to launch a new subtask. (CIEL also allows subtasks to spawn new subtasks, but requires use of its Skywriting programming language to do this.)

Burst-parallel cloud functions. Researchers and practitioners have taken advantage of cloud-functions platforms to implement low-latency, massively parallel applications. ExCamera [15] uses AWS Lambda to scale out video encoding and processing tasks over thousands of function invocations, and PyWren [23] exposes a MapReduce-like Python API that executes on AWS Lambda. Spark-on-Lambda [40] is a port of Spark that uses AWS Lambda cloud functions. In contrast, gg helps applications use cloud-functions platforms for a broader set of workloads, including irregular execution graphs and ones that change as execution evolves. gg's main contribution is specifying an IR that permits a diverse class of applications (written in any programming language) to be abstracted from the compute and storage platform, and to leverage common services for dependency management, straggler mitigation, and scheduling.

Build tools. Several build systems (e.g., `make` [13], Bazel [4], Nix [11], and Vesta [19]) and outsourcing tools (such as `distcc` [8], `icecc` [20], and `mrcc` [26]) seek to incrementalize, parallelize, or distribute compilation to more-powerful

remote machines. Building on such systems, gg automatically transforms existing build processes into their own IR. The goal is to compile programs quickly—irrespective of the software's own build system—by making use of cloud-functions platforms that can burst from complete dormancy to thousands-way parallelism and back.

Existing remote compilation systems, including `distcc` and `icecc`, send data between a master node and the workers frequently during the build. These systems perform best on a local network, and add substantial latency when building on more remote servers in the cloud. In contrast, gg uploads all the build input once and executes and exchanges data purely within the cloud, reducing the effects of network latency.

3 Design and Implementation

gg is designed as a general system to help application developers manage the challenges of creating burst-parallel cloud-functions applications. The expectation is that users will take computations that might normally run locally or on small clusters for a long time (e.g., test suites, machine learning, data exploration and analysis, software compilation, video encoding and processing), and outsource them to thousands of short-lived parallel threads in the cloud, in order to achieve near-interactive completion time.

In this section, we describe the design of gg's intermediate representation (§3.1), front-end code generators (§3.2), and back-end execution engines (§3.3).

3.1 gg's Intermediate Representation

The format that gg uses—a set of documents describing a container and its dependency on other containers—is intended to elicit enough information from applications about their jobs (fine-grained dependencies and dataflow) to be able to efficiently execute a job on constrained and stateless cloud functions. It includes:

1. A primitive of a content-addressed cloud *thunk*: a codelet or executable applied to named input data.
2. An intermediate representation (IR) that expresses jobs as a lazily evaluated lambda expression of interdependent thunks.
3. A strategy for representing dynamic computation graphs and data-access patterns in a language-agnostic and memoizable way, using tail recursion.

We discuss each of these elements.

3.1.1 Thunk: A Lightweight Container

In the functional-programming literature, a thunk is a parameterless closure (a function) that captures a snapshot of its arguments and environment for later evaluation. The process of evaluating the thunk—applying the function to its arguments and saving the result—is called *forcing* it [2].

For gg, our goal is to simplify the creation of new applications by allowing them to target the IR, which lets them leverage the common services provided by the back-end engines. Accordingly, the representation of a thunk follows from several design goals. It should be: (1) simple enough to be portable to different compute and storage platforms, (2) general enough to express a variety of plausible applications, (3) agnostic to the programming language used to implement the function, (4) efficient enough to capture fine-grained dependencies that can be materialized on stateless and space-limited cloud functions, and (5) able to be memoized to prevent redundant work.

To satisfy these requirements, gg represents a thunk with a description of a container that identifies, in content-addressed manner, an x86-64 Linux executable and all of its input data objects. The container is hermetically sealed: it is not allowed to use the network or access unlisted objects or files. The thunk also describes the arguments and environment for the executable, and a list of tagged output files that it will generate—the results of forcing the thunk. The thunk is represented as a Protobuf [31] structure (Figure 3 shows three thunks for three different stages of a build process). This container-description format is simple to implement and reason about, and is well-matched to the statelessness and unreliability of cloud functions.

In the content-addressing scheme, the name of an object has four components: (1) whether the object is a primitive value (hash starting with `V`) or represents the result of forcing some other thunk (hash starting with `T`), (2) a SHA-256 hash, (3) the length in bytes, and (4) an optional tag that names an object or a thunk's output.

Forcing a thunk means instantiating the described container and running the code. To do this, the executor must fetch the code and data values. Because these are content-addressed, this can be from any mechanism capable of producing a blob that has the correct name—durable or ephemeral storage (e.g., S3, Redis, or Bigtable), a network transfer from another node, or by finding the object already available in RAM from a previous execution. The executor then runs the executable with the provided arguments and environment—for debugging or security purposes, preferably in a mode that prevents the executable from accessing the network or any data not listed as a dependency. The executor collects the output blobs, calculates their hashes, and records that the outputs can be substituted in place of any reference to the just-forced thunk.

3.1.2 gg IR: A Lazily Evaluated Lambda Expression

The structure of interdependent thunks is what defines the gg IR. We use a one-way IR, a document format that applications write to express their jobs, as opposed to a two-way API (e.g., a function call to spawn a new task and observe its result)

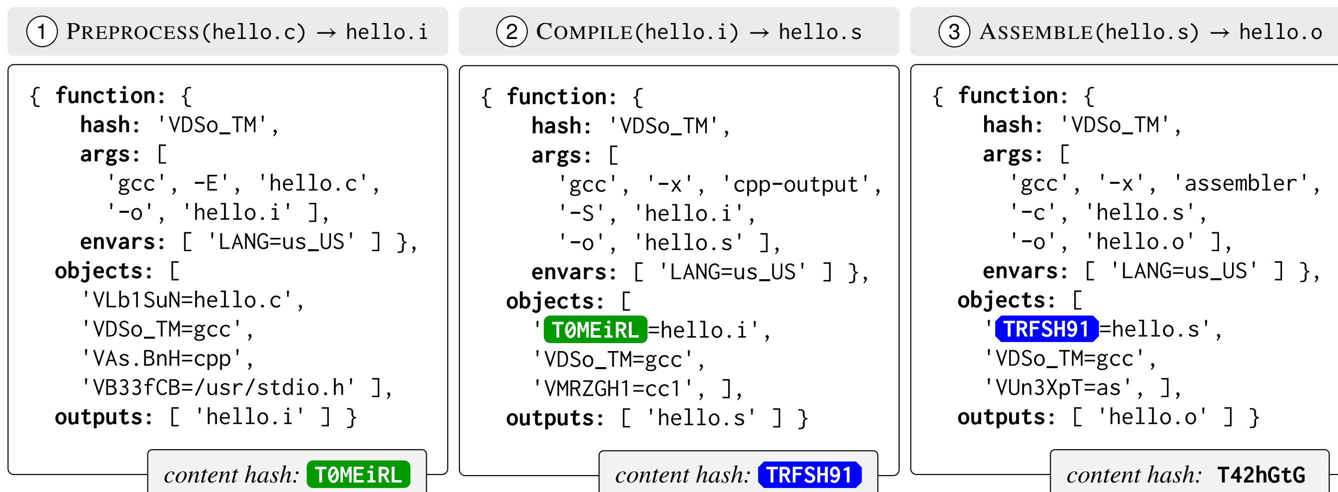


Figure 3: An example of gg IR consisting of three thunks for building a "Hello, World!" program that represents the expression ASSEMBLE(COMPILE(PREPROCESS(hello.c))) → hello.o. To produce the final output hello.o, thunks must be forced in order from left to right. Other thunks, such as the link operation, can reference the last thunk's output using its hash, T42hGtG. Hashes have been shortened for display, and dependencies between thunks are shown in color.

because we expect the application will be running on the user's own computer, at some remote cloud-functions engine: the intention is to avoid roundtrips over a long-latency path by keeping the application out of the loop. We also envision that it will be possible to better schedule and optimize a job, and easier to maintain different interoperable back-ends, if the application is out of the loop before execution begins.[2] This representation exposes the computation graph to the back-end, along with the identities and sizes of objects that need to be communicated between thunks. Based on this information, the back-end can schedule the forcing of thunks, place thunks with similar data-dependencies or an output-input relationship on the same physical infrastructure, and manage the storage or transfer of intermediate results, without roundtrips back to the user's own computer.

The IR allows gg to schedule jobs efficiently, mitigate the effect of stragglers by invoking multiple concurrent thunks on the critical path, recover from failures by forcing a thunk a second time, and memoize thunks. This is achieved in an application-agnostic, language-agnostic manner.

The application generally starts by forcing a single thunk that represents the ultimate outcome of the interactive operation. This thunk typically depends on other thunks that need to be forced first, etc., leading the back-end to lazily force thunks recursively until obtaining the final result. Figure 3 shows an example IR for computing the expression ASSEMBLE(COMPILE(PREPROCESS(hello.c))).

3.1.3 Tail Recursion: Supporting Dynamic Execution

The above design is sufficient to describe a directed acyclic graph (DAG) of deterministic tasks executing in the cloud. However, many jobs do not have a data-access pattern that is completely known upfront. For example, in compiling software, it is unknown *a priori* which header files and libraries will need to be read by a given stage. Other applications use loops, recursion, and other non-DAG dataflows.

An application may also have an unpredictable degree of parallelism. For example, an application might detect objects in a large image, and then on each subregion where an object is detected (which may be zero regions, or might be 10,000 regions), the application searches for a target object. Here, the computation graph is not known in advance.

Systems like PyWren [23] and CIEL's Skywriting language [27] handle this case by giving tasks access to an API call to invoke a new task. For gg, we aimed to preserve the memoizability and language-independence of the IR, which is challenging if tasks can invoke tasks on their own and if gg must expose a language binding. Instead, gg handles this situation through language-independent tail recursion: a thunk can write another thunk as its output.

3.2 Front-ends

We developed four front-ends that emit gg IR: a C++ SDK, a Python SDK, a group of command-line tools, and a series

[2] Systems like the LLVM compiler suite [25] (which allows front-end language compilers to benefit from a library of back-end optimization passes and assemblers, interfacing through an IR) and Halide [33] (which separates an image-processing algorithm from its schedule and execution strategy) have demonstrated the benefits of a rigid representational abstraction in other settings. gg's use of an IR is not exactly the same as these, but it has a similar value in abstracting front-ends (applications and the tools that help them express their jobs) from back-end execution engines in a way that allows efficient and portable execution.

of *model substitution* primitives that can infer gg IR from a software build system.

The C++ and Python SDKs are straightforward. Each exposes a thunk abstraction and allows the developer to describe a parallel application in terms of codelets. These codelets are applied to blobs of named data, which may be read-only memory regions or files in the filesystem.

The model-substitution primitives extract a gg IR description of an existing build system, without actually compiling the software. Instead, we run the build system with a modified PATH so that each stage is replaced with a stub: a *model* program that understands the behavior of the underlying stage well enough so that when the model is invoked in place of the real stage, it can write out a thunk that captures the arguments and data that will be needed in the future, so that forcing the thunk will produce the exact output that would have been produced during actual execution. We used this technique to infer gg IR from the existing build systems for several large open-source applications (§4.1).

3.3 Back-ends

gg IR express the application against an abstract machine that requires two components: an *execution engine* for forcing the individual thunks, and a content-addressed *storage engine* for storing the named blobs referenced or produced by the thunks. The *coordinator* program brings these two components together.

Storage engine. A storage engine provides a simple interface to a content-address storage, consisted of GET and PUT functions to retrieve and store objects. We implemented several content-addressed storage engines, backed by S3, Redis, and Google Cloud Storage. We also have a preliminary implementation (not evaluated here) that allows cloud functions to communicate directly among one another, avoiding the latency and throughput limitations of using a reliable blob storage (e.g., S3) to exchange small objects.

Execution engine. In conjunction with a storage engine, each execution engine implements a simple abstraction: a function that receives a thunk as the input and returns the hashes of its output objects (which can be either values or thunks). The engine can execute the thunk *anywhere*, as long as it returns correct output hashes that are retrievable from the storage engine. We implemented back-end execution engines for several environments: a local multicore machine, a cluster of remote VMs, AWS Lambda, Google Cloud Functions, and IBM Cloud Functions (OpenWhisk).

The coordinator. The main entry-point for executing a thunk is the *coordinator* program. The inputs to this program are the target thunk, a list of available execution engines and the storage engine. This program implements services offered by gg, such as job scheduling, memoization, failure recovery and straggler mitigation.

Upon start, this program materializes the target thunk's dependency graph, which includes all the other thunks needed to get the output. Then, the thunks that are ready to execute are passed to execution engines, based on their available capacity. When the execution of a thunk is done, the program updates the graph by replacing the references to the just-forced thunk and adds a cache entry associating the output hash to the input hash. The thunks that become ready to execute are placed on a queue and passed to the execution engines when their capacity permits. The unified interface allows the user to mix-and-match different execution engines, as long as they share the same storage engine.

The details of invocation, execution and placement are left to the execution engines. For example, the default engine for AWS Lambda/S3 invokes a new Lambda for each thunk. The Lambda downloads all the dependencies from S3 and sets up the environment, executes the thunk, uploads the outputs back to S3 and shuts down. For applications with large input/output objects, the roundtrips to S3 could affect the performance. As an optimization for such cases, the user can decide to run the execution engine in the "long-lived" mode, where each Lambda worker stays up until the job finishes and seeks out new thunks to execute. The execution engine keeps an index of all the objects that are already present on each worker's local storage. When placing thunks on workers, it selects the worker with the most data available, in order to minimize the need to fetch dependencies from the storage back-end.

The coordinator can also apply optimizations to the dependency graph. For example, multiple thunks can be bundled as one and sent to the execution engine. This is useful when the output of one thunk will be consumed by the next thunk, creating a linear pipeline of work. By scheduling all of those thunks on one worker, the system reduces the number of roundtrips.

Failure recovery and straggler mitigation. In case of coordinator failure, the job can be picked up where it was left off, as the coordinator program uses on-disk cache entries to avoid redoing the work that has already been done. In case of a recoverable error in executing a thunk, the execution engine notifies the coordinator with the failure reason, where it can decide to retry the job or pass it to another available execution engine for execution.

Straggler mitigation is another service managed by the coordinator program which duplicates pending executions in the same or a different execution engine. The program keeps track of the execution time for each thunk, and if the execution time exceeds a timeout (set by either the user or the application developer) the job will be duplicated. Since the functions don't have any side-effects, the coordinator simply picks the output that becomes ready first.

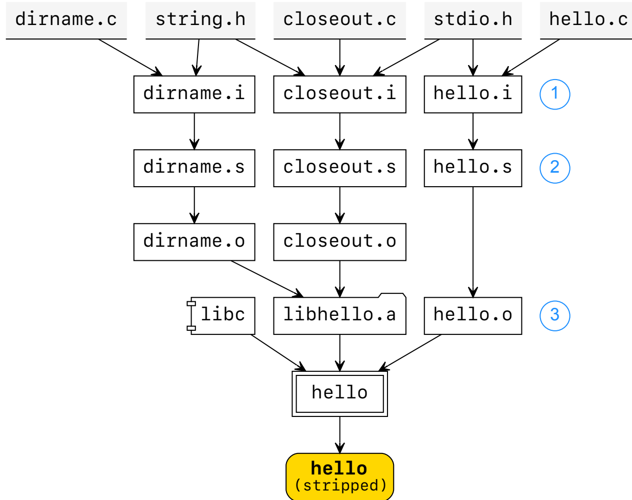


Figure 4: Part of the IR of interdependent thunks inferred with model substitution from the GNU `hello` build system. Each box represents a thunk and is labeled with the name of its output. The contents of the numbered thunks are depicted in Figure 3 (Many header files and other dependencies omitted for simplicity).

3.4 Implementation Notes

We implemented gg in about 14,000 lines of C++. The implementation consists of five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, Google Cloud Functions, and IBM Cloud Functions), three storage engines (S3, Google Cloud Storage, and Redis), a series of command line tools to aid generation, execution and inspection of gg IR, a C++ and Python SDK, and several model programs for different stages of build process.

4 Applications

We used gg to implement several applications, each emitting jobs in the gg IR. We describe these in turn.

4.1 Software Compilation

The time required to compile software is an evergreen frustration for software developers; a popular cartoon even spoofs the duration of this job [39]. Today's open-source applications have grown larger and larger. For example, the Chromium Web browser takes more than four *hours* to compile on a four-core laptop computer from a cold start. Many solutions have been developed to leverage warm machines in a local cluster or cloud datacenter (e.g., `distcc` or `icecc`). We developed such an application on top of gg.

Using model substitution, we implemented models for seven popular stages of a C or C++ software build pipeline: the preprocessor, compiler, assembler, linker, archiver, indexer, and `strip`. These allow us to automatically transform some software build processes (e.g., a `Makefile` or `build.ninja` file) into an expression in gg IR, which can then be executed with thousands-way parallelism on cloud-functions platforms to obtain the same results as if the build system had been executed locally. Figure 4 illustrates the resulting IR from an example invocation (the enumerated thunks are detailed in Figure 3). These models are sufficient to capture the build process of some major open-source applications, including OpenSSH [29], Python interpreter [32], the Protobuf library [31], the FFmpeg video system [14], the GIMP image editor [16], the Inkscape vector graphics editor [21], and the Chromium browser [6].[3]

Build systems often include scripts that run in addition to these standard tools, such as a tool to generate configuration header files, but typically such scripts run upstream of the preprocessor, compiler, etc. Therefore, gg captures these script outputs by a model as dependencies.

Capturing dependencies of the preprocessor. Preprocessing is the most challenging stage to model. It requires not only capturing the source file as dependencies, but also all the header files that are both directly and indirectly included by that source file. Capturing *all* header files in a container is not feasible, because cloud functions are constrained in storage. For example, AWS Lambda has a 500 MB storage limit.

The precise header files required to preprocess a file can be discovered at fine grain, but only by invoking the preprocessor (i.e., `gcc -M`) which is an expensive operation at large scale. Finding the dependencies for each source file in Chromium takes nearly half an hour on a 4-core computer.

To solve this problem, the application uses gg's capabilities for dynamic dataflow at runtime. gg's preprocessor model generates thunks that do dependency inference in parallel on cloud functions. These thunks have access only to a stripped-down version of the user's include directories, preserving only lines with C preprocessor directives (such as `#include` and `#define`). These thunks then produce further thunks that preprocess a given source-code file by listing only the necessary header files.

4.2 Unit Testing

Software test suites are another application that can benefit from massive parallelism. Using gg's C++ SDK, we implemented a tool that can generate gg IR for unit tests written with Google Test [17], a popular C++ test framework used by projects like LLVM, OpenCV, Chromium, Protocol Buffers, and the VPX video codec library.

[3] We have to emphasize that no changes were made to the underlying build system of these programs. The main challenge here is to build correct and complete models for programs used in the build pipeline, such as `gcc` and `ld`, which is a one-time effort. However, an arbitrary build system may require other programs to be modeled, or execute these programs in an aberrant way that is outside of the scope of model substitution.

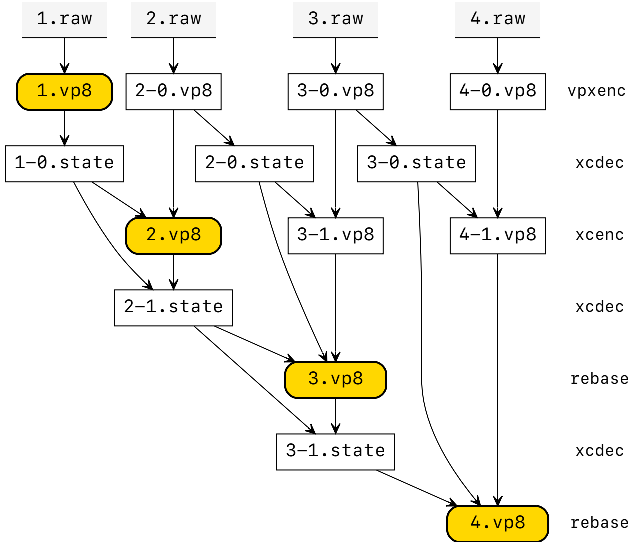


Figure 5: Visual representation of the gg IR for a video-processing workflow [15].

Typically, each test is a standalone program that can be run in parallel with other tests, with no dependency requirements between them. No changes to the code are necessary, with one exception: if a test case needs to access files on the file system, then the programmer has to annotate the test case with the list of files that it wants to access. This process can be automated by running the tests locally and then tracing the open system calls invoked by each test case. The tool uses these annotations, either handcrafted or automatically generated, to capture each test's dependencies. A separate thunk is created for each test case, allowing the execution engine to exploit the available parallelism.

4.3 Video Encoding

The ExCamera system [15] uses cloud-functions infrastructure to run interdependent video-processing tasks with 4,000-way parallelism [15]. Cloud workers exchange data through TCP connections, brokered by a tightly coupled back-end that was bound to AWS Lambda. To demonstrate gg's expressive power and performance, we ported ExCamera into a "front-end-only" version that targets gg IR.

In ExCamera, the functions necessary for parallel video encoding are ENCODE, DECODE, ENCODE-GIVEN-STATE, and REBASE. The algorithm first encodes each chunk in parallel using ENCODE and then, in a serial process, REBASEs each output on top of the state left by the previous chunk. Video-codec states must be communicated between workers in order to stitch together the overall video. Figure 5 shows the dependency graph for encoding a batch of four chunks.

The original ExCamera keeps Lambda workers warm by keeping the raw video in RAM and communicating video-codec states over TCP between workers. gg's back-end for AWS Lambda also keeps workers warm and keeps the raw video in their local filesystem. gg routes thunks to workers that already have the necessary data, but brokers inter-worker communication through S3. Finally, gg provides fault tolerance, which ExCamera's own back-end lacks.

4.4 Object Recognition

The increase in visual computing applications has motivated the design of frameworks such as Scanner [30], which is a system for productive and efficient video analysis at scale. To use Scanner, the user feeds in a compressed video and designates an operation to be applied on each decoded frame. To compare Scanner's execution engine with gg, we used the gg C++ SDK to implement a two-step analysis pipeline. In the first stage, the frames of a video V are decoded in batches of m frames, using DECODE(V,m) function. Subsequently, an object-recognition kernel, OBJECT-REC, is applied to the decoded frames and returns the top five recognized objects for each frame.

We implemented the DECODE function using FFmpeg [14] and implemented OBJECT-REC in TensorFlow's C++ API [1] using a pre-trained Inception-v3 model [37]. gg's thunks were able to bundle these pre-existing applications. We implemented the same pipeline in Scanner for comparison. To do so, we leverage Scanner's internal video decoder and the same TensorFlow kernel and pre-trained Inception-v3 model.

4.5 Recursive Fibonacci

To demonstrate the way that gg handles dynamic dataflows, we used the C++ SDK to implement a classic recursive Fibonacci program in the gg IR. The application is expressed using two functions: ADD(a,b), which returns the sum of its two input values and FIB(n) which recursively computes the n-th Fibonacci number as ADD(FIB($n-1$), FIB($n-2$)) or the base case when $n \leq 1$.

Figure 6 shows the execution steps. In the beginning, there is only one thunk, FIB(4). After execution, instead of returning a value, it returns three thunks, replacing the target with the sum of two preceding Fibonacci numbers. The IR expands (for the recursive case) and contracts (for the base case), until resolving to the final value.

In a naïve recursive implementation of the Fibonacci series, each Fibonacci value is evaluated many times. However, in gg, the functions are memoized and lazily-executed, resulting in each Fibonacci value computed only once.

5 Evaluation

We evaluated gg's performance by executing each application in gg, compared with comparable tailor-made or native applications. Although we implemented back-end engines for

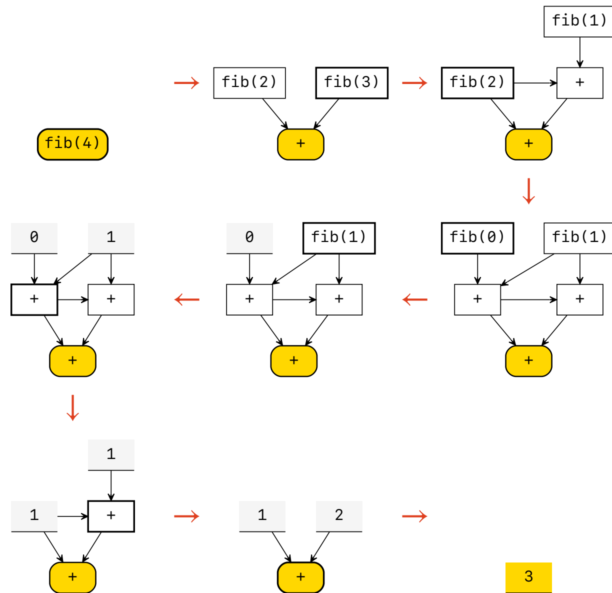


Figure 6: Evolution of the IR for a recursive Fibonacci application. Execution begins with a single thunk. As each thunk is forced, returning a new thunk or the base case, the IR expands and contracts. The engine lazily forces thunks until it can return the overall value.

1K trivial containers running "sleep 2"		
AWS Lambda	gg-λ	06s ± 01s
	PyWren	46s ± 08s
	Spark-on-Lambda	54s ± 21s
Google Kubernetes Engine	Kubernetes	03m 08s ± 03s

Figure 7: Comparison of completion time for running 1,000 `sleep(2)` tasks using four different systems. gg's lightweight design and implementation has less overhead than other systems.

Compiling Inkscape on AWS Lambda (total of 3602 thunks)	
Initial graph construction	56 ms
Mean time to read a thunk	188 µs ± 367 µs
Mean time to recompute the IR per thunk	336 µs ± 560 µs
Invocation (thunk completion to invocation of all dependent thunks)	142 ms ± 135 ms

Figure 8: gg's overheads allow for relatively fine-grained tasks.

several cloud-functions platforms (including Google Cloud Functions and IBM Cloud Functions), we found that AWS Lambda had the best performance and available parallelism. As a result, we focus on evaluation results from gg's AWS Lambda back-end.

5.1 Startup Overhead

To motivate the importance of gg's lightweight abstractions, we implemented a trivial job, 1,000 parallel tasks each executing `sleep(2)`, using four frameworks: gg, PyWren, Spark-on-Lambda, and Kubernetes. The first three frameworks were executed on AWS Lambda, and the last on Google Kubernetes Engine (GKE), which was given a warm cluster of eleven 96-core VMs (1,056 cores in total) on which to allocate containers. Figure 7 shows the results.

gg is able to quickly scale to 1,000 concurrent containers and finish the job 7.5–9× faster when compared with other two frameworks running on Lambda. After subtracting off the 2-second sleep time, this translates to 11–13× less overhead. For PyWren, on average, each worker spends 70% of its time on setting up the Python runtime (downloading and extracting a tarball). A large portion of this runtime consists of packages that are not used by our `sleep(2)` program (cf. gg fine-grained dependency tracking). Google Kubernetes Engine was not designed for transient computations and was not optimized for this use case; it is much slower to start 1,000 Docker containers.

Additionally, we measured the overheads associated with loading thunks and recomputing the IR after a thunk is done in Figure 8. These overheads, especially the invocation overhead, support an intuition about the appropriate granularity of thunks: gg works well when thunks last about 1–20 seconds each.

5.2 Software Compilation

To evaluate gg's application for software compilation, we measured the start-to-finish build times under multiple scenarios on a set of unmodified large open-source packages. We compared these times with existing tools under the same scenarios. For distributed builds outsourced from a 4-core EC2 VM, we found that gg is able to achieve significantly shorter build times than existing approaches.

5.2.1 Evaluation Set

To benchmark gg's performance, we picked four open-source programs written in C or C++: FFmpeg, GIMP, Inkscape, and Chromium. No changes were made to the code or the underlying build system of these packages. We compiled all packages with GCC 7.2.

All the 4-core machines used in the experiments are EC2 `m5.xlarge`, and all the 48-core machines are EC2 `m5.12xlarge` instances. To realistically simulate users sending applications to nearby datacenters, client machines reside in the US West (N. California) region, and outsource their jobs to machines in the US West (Oregon) region.

	Estimated SLoC	Local (make) 1 core	Local (make) 48 cores	Distributed (icecc) 48 cores	Distributed (icecc) 384 cores	**Distributed (gg)** 384 cores	**Distributed (gg)** AWS Lambda
FFmpeg	1,200,000	06m 19s	20s	01m 03s	39s	40s	44s ± 04s
GIMP	800,000	06m 48s	49s	02m 35s	02m 38s	01m 26s	01m 38s ± 03s
Inkscape	600,000	32m 34s	01m 40s	06m 51s	06m 57s	01m 20s	01m 27s ± 07s
Chromium	24,000,000	15h 58m 20s	38m 11s	46m 01s	42m 18s	40m 57s	18m 55s ± 10s

Figure 9: Comparison of cold-cache build times in different scenarios described in §5.2. gg on AWS Lambda is competitive with or faster than using conventional outsourcing (icecc), and in the case of the largest programs, 2–5× faster. This includes both the time required to generate gg IR from a given repository and then to execute the IR.

5.2.2 Baselines

For each package, we measured the start to finish build time in four different scenarios as the baseline for local and distributed builds:

make, make (48): The package's own build system was executed on a single core (make), and with up to 48-way parallelism (make -j48). The make and make (48) tests were done on 4-core and 48-core EC2 VMs, respectively. No remote machines were involved in these tests.

icecc (48), icecc (384): The package was built using the icecc distributed compiler on a 4-core client that outsources the job to a 48-core VM, or to eight 48-core VMs, for a total of 384 cores.

5.2.3 gg's Benchmarks

We conducted the following experiments for each package to evaluate gg:

1. **gg (384)**: The package was built with the same configuration as the icecc (384) experiment: a 4-core client farming out to eight 48-core machines, using gg's backend for a cluster of VMs.
2. **gg-λ**: The package was built on a 4-core client outsourcing to AWS Lambda, using as many concurrent Lambdas as possible (up to 8,000 in the case of Chromium).

   For Chromium experiments, an additional standby EC2 VM acted as the *overflow* worker for thunks whose total data size exceeded Lambda's storage limit of 500 MB. Throughout building Chromium, there were only 2 thunks (out of ~90,000 thunks) that did not fit on a Lambda and had to be forced on this overflow node.

5.2.4 Discussion of Evaluation Results

Figure 9 shows the median times for the package builds. gg is about 2–5× faster than a conventional tool (icecc) in building medium- and large-sized software packages. For example, gg compiles Inkscape in 87 seconds on AWS Lambda, compared with 7 minutes when outsourced with icecc to a warm 384-core cluster. This is a 4.8× speedup. Chromium, one of the largest open-source projects available, compiles in under 20 minutes using gg on AWS Lambda, which is 2.2× faster than icecc (384).

We do not think gg's performance improvements on AWS Lambda can be explained simply by the availability of more cores than our 384-core cluster; icecc improved only modestly between the 48-core and 384-core case and doesn't appear to effectively use higher degrees of parallelism. This is largely because icecc, in order to simplify dependency tracking, runs the preprocessor locally, which becomes a major bottleneck. gg's fine-grained dependency tracking allows the system to efficiently outsource this step to the cloud and minimize the work done on the local machine.

Figure 10 shows an execution breakdown for compiling Inkscape. We observe two important characteristics. First, the large spikes correspond to Lambdas that have failed or taken longer than usual to complete. gg's straggler mitigation detects and relaunches these jobs to prevent an increase in end-to-end latency. Second, the last few jobs are primarily serial (archiving and linking), and consume almost a quarter of the total job-completion time. These characteristics were also observed in the other build jobs.

5.3 Unit Tests

To benchmark gg's performance in running unit tests created with the Google Test framework, we chose the VPX video codec library [9], which contains ~7,000 unit tests. We annotated each test with the list of required data files.

The Google Test library that is shipped with LibVPX is only capable of running the tests serially. To establish a better baseline, we used gtest-parallel, a program that executes Google Test binaries in parallel on the local machine. We ran the tests with 4- and 48-way parallelism and compared the results with gg on AWS Lambda, with 8,000-way parallelism. Figure 11 shows the summary of these results.

Using the massive parallelism available, gg was able to execute all of the test cases in parallel, and 99% of the test

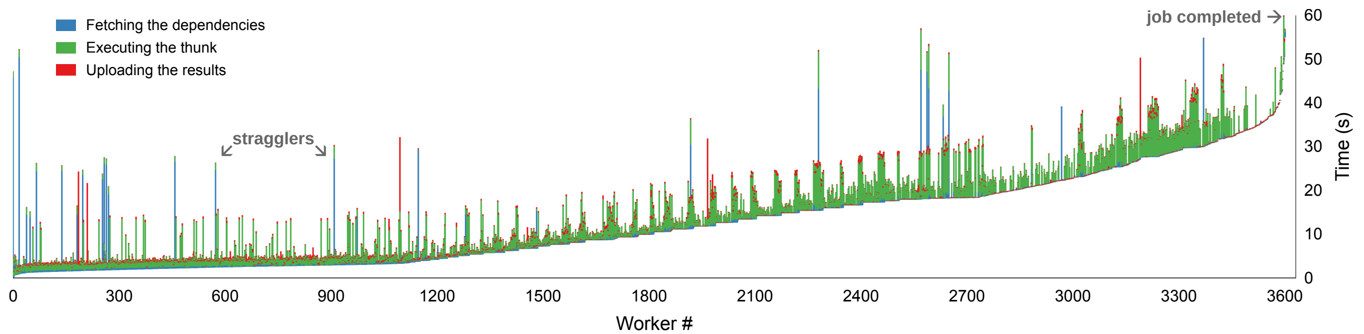


Figure 10: Breakdown of workers' execution time when building Inkscape using gg on AWS Lambda. Serial stages (archiving and linking) consume almost a quarter of the total job-completion time. Spikes indicate stragglers, which are mitigated by gg using standard techniques. In this experiment, stragglers mostly consist of Lambdas that have trouble communicating with the storage back-end (S3).

	gg-λ	gtest (4)	gtest (48)
LibVPX Test Suite	03m 25s	51m 45s	04m 40s

Figure 11: Running the LibVPX test suite using gg on AWS Lambda outperforms running the tests with 4-way and 48-way parallelism on a local machine. 99% of the test cases complete within 30 seconds.

	gg-λ	original
ExCamera	01m 30s	01m 16s

Figure 12: The gg version of ExCamera is 18% slower than the hand-optimized original ExCamera, which was written to pipeline I/O and computation within a Lambda worker.

cases finished within the first 30 seconds. From a developer's point of view, this improves turnaround time and translates into faster discovery of bugs and regressions.

5.4 Video Encoding

We evaluated the gg implementation of ExCamera on AWS Lambda, with the original implementation as the baseline. The selected configuration was the same as ExCamera's original paper (6 frames per chunk, 16 chunks per batch). The input video consisted of 888 chunks, and all chunks had been uploaded to S3 in raw format prior to execution. For the original ExCamera implementation, a 64-core VM (`m4.16xlarge`) was used as the rendezvous server to broker TCP streams between Lambda workers.

Figure 12 shows the results. The original ExCamera was hand-coded to pipeline I/O and computation to reduce end-to-end latency. By contrast, gg's abstract interface must force and load all data-dependencies before running user code, and cannot perform this optimization. ExCamera-on-gg is 18% slower than the original, but adds memoization and fault-tolerance, unlike the original ExCamera.

Object Recognition	
gg local (64 cores)	04m 30s
gg on AWS Lambda	37s
Scanner local (64 cores)	05m 39s
Scanner on cluster (140 cores)	03m 14s

Figure 13: Scanner-on-gg outperforms the original Scanner on the same hardware, and performs even faster on AWS Lambda.

5.5 Object Recognition

We compared the original Scanner [30] with the gg implementation using a 4K video with more than 6,000 frames. For the baseline, we chose the most favorable execution parameters through an exhaustive search. The optimal number of pipeline instances and frame batches were 14 and 75, respectively. Within each pipeline, each video chunk is first decoded into raw images before being passed to the TensorFlow kernel execution thread. Each execution thread only needs to load the model once per stream of frames. Scanner local was run on a 64-core machine (`m4.16xlarge`). Scanner on cluster was run with a 4-core master (`m4.xlarge`) and four 36-core workers (`c4.8xlarge`), of which Scanner uses 35 and leaves one for scheduling. For the gg implementation, the video was broken up into five-second chunks and uploaded to the cloud prior to execution. Each chunk was decoded in batches of 25 frames. For the object recognition task, the IR was configured to the optimal number of frame batches per task.

Figure 13 presents the summary of the results. While Scanner on cluster is 39% faster than gg local, it is 5.2× slower than gg on AWS Lambda. Scanner local is over 9× slower than gg on AWS Lambda. gg's lightweight scheduling and execution engine removes several layers of abstraction present in Scanner's design.

6 Limitations and Discussion

gg has a number of important limitations and opportunities for future work.

Direct communication between workers. Although commentators have noted that "*two Lambda functions can only communicate through an autoscaling intermediary service... like S3*" [18], our experience differs: we have found that on AWS Lambda, two Lambda functions can communicate directly using off-the-shelf NAT-traversal techniques, at speeds up to 600 Mbps (although the performance is variable and requires an appropriate protocol and failure-recovery strategy). We thus believe that the performance of systems such as ExCamera, PyWren, and gg is likely to improve in the future as practitioners develop better mechanisms for harnessing this computing substrate, including direct communication.

In follow-on work, we are developing a 3D ray-tracing engine on gg, that will quickly render complex scenes across thousands of nodes, where the scene geometry and textures consume far more space than any individual node's memory. To achieve sufficient performance, this will require low-latency and high-speed communication between workers, motivating the use of direct network connectivity, instead of an intermediate storage system such as S3 or Pocket [24].

Limited to CPU programs. gg specifies the format of the code as an x86-64 Linux ELF executable. The IR has no mechanism to signal a need for GPUs or other accelerators, and efficiently scheduling such resources poses nontrivial challenges, because loading and unloading configuration state from a GPU is a more expensive operation than memory-mapping a file. We plan to investigate the appropriate mechanisms for a gg back-end to schedule thunks onto GPUs.

A gg DSL to program for the IR. Currently, we have implemented a C++ and Python SDK for users to express applications that target the gg IR. However, this requires the user to explicitly provide an x86-64 executable and all of its dependencies prior to thunk generation. We envision a language in which users can write high-level code in Python or C++, using primitives such as a parallel map, fold, and other operations, which will be compiled into the gg IR.

Why cloud functions? Transient, burst-parallel execution on services like AWS Lambda produces a different cost structure from a warm cluster. It takes about the same amount of time for gg to compile Inkscape on AWS Lambda as on a 384-core cluster of warm EC2 VMs (Figure 9). The job costs about 50 cents *per run* on Lambda, compared with $18.40 *per hour* to keep a 384-core cluster running (Figure 2). Whether it is financially beneficial for the gg user to run such jobs on long-running VMs or on cloud functions depends on how often the user has a job to run. From an economic perspective, the provider is compensating the infrequent user for their elasticity; e.g., for having structured their workload to vacate compute resources when no task is active, and to tolerate variations in the exact number of nodes available for a job and the timing of when they are allocated.

In the future, we expect the performance characteristics of VMs and Lambda-like services to move closer together. There is no intrinsic reason for it to take more than 30 seconds to provision and boot an infrastructure-as-a-service VM in the public cloud. Linux itself can boot in less than a second, and KVM and VMware can provision a VM in less than 3 seconds. We understand the remaining time is largely "management plane" overhead. If this can be reduced, then cloud functions may hold no compelling advantage over virtual machines for executing burst-parallel applications—but tools like gg that aid efficient execution on remote compute infrastructure (whether VM or cloud function) may remain valuable.

7 Conclusion

In this paper, we described gg, a framework that helps developers build and execute burst-parallel applications. gg presents a portable abstraction: an intermediate representation (IR) that captures the future execution of a job as a composition of lightweight Linux containers. This lets gg support new and existing applications in various languages that are abstracted from the compute and storage platform and from runtime features that address underlying challenges: dependency management, straggler mitigation, placement, and memoization.

As a computing substrate, we suspect cloud functions are in a similar position to Graphics Processing Units in the 2000s. At the time, GPUs were designed solely for 3D graphics, but the community gradually recognized that they had become programmable enough to execute some parallel algorithms unrelated to graphics. Over time, this "general-purpose GPU" (GPGPU) movement created systems-support technologies and became a major use of GPUs, especially for physical simulations and deep neural networks.

Cloud functions may tell a similar story. Although intended for asynchronous microservices, we believe that with sufficient effort by this community the same infrastructure is capable of broad and exciting new applications. Just as GPGPU computing did a decade ago, nontraditional "serverless" computing may have far-reaching effects.

Acknowledgments

We thank the USENIX ATC reviewers and our shepherd, Ed Nightingale, for their helpful comments and suggestions. We are grateful to Geoffrey Voelker, George Porter, Anirudh Sivaraman, Zakir Durumeric, Riad S. Wahby, Liz Izhikevich, and Deepti Raghavan for comments on versions of this paper. We also thank Alex Ozdemir for his measurements on Lambda networking, and Alex Poms for all his help with Scanner. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-2-0047, and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] 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 http://tensorflow.org.

[2] Harold Abelson and Julie Sussman, G. J. with Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *ACM Symposium on Cloud Computing (SoCC 2018)*, Carlsbad, CA, 2018.

[4] Bazel build system. https://bazel.build.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):10:70–10:93, January 2016.

[6] The Chromium browser. https://www.chromium.org/Home.

[7] CMake. https://cmake.org.

[8] distcc distributed compiler. https://github.com/distcc/distcc.

[9] LibVPX: Vp8/vp9 codec sdk. https://www.webmproject.org/code/.

[10] Docker. https://www.docker.org.

[11] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006.

[12] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. The Utility Coprocessor: Massively parallel computation from the coffee shop. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[13] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 9(4):255–65, 1979.

[14] FFmpeg. https://github.com/FFmpeg/FFmpeg.

[15] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[16] GIMP. https://www.gimp.org/.

[17] Google Test — Google Testing and Mocking Framework. https://github.com/google/googletest.

[18] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[19] Clark Allan Heydon, Roy Levin, Timothy P. Mann, and Yuan Yu. *Software Configuration Management Using Vesta*. Springer Publishing Company, Incorporated, 2011.

[20] Icecream distributed compiler. https://github.com/icecc/icecream.

[21] Inkscape, a powerful, free design tool. https://inkscape.org.

[22] 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.

[23] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

[24] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[25] The LLVM compiler infrastructure. http://llvm.org.

[26] Zhiqiang Ma and Lin Gu. The limitation of MapReduce: A probing case and a lightweight solution. In *Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization*, pages 68–73, 2010.

[27] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, 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.

[28] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[29] OpenSSH. https://www.openssh.com.

[30] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. In *ACM Transactions on Graphics*, 2018. Software available from https://github.com/scanner-research/scanner.

[31] Protocol Buffers. https://github.com/google/protobuf.

[32] Python. https://www.python.org.

[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. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[34] George Sammons. *Learning Vagrant: Fast Programming Guide*. CreateSpace Independent Publishing Platform, USA, 2016.

[35] Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/.

[36] Apache Spark on AWS Lambda. https://github.com/qubole/spark-on-lambda.

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[38] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[39] xkcd — Compiling. https://xkcd.com/303/.

[40] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries

Mohammad Hedayati
University of Rochester

Spyridoula Gravani
University of Rochester

Ethan Johnson
University of Rochester

John Criswell
University of Rochester

Michael L. Scott
University of Rochester

Kai Shen
Google

Mike Marty
Google

Abstract

As network, I/O, accelerator, and NVM devices capable of a million operations per second make their way into data centers, the software stack managing such devices has been shifting from implementations within the operating system kernel to more specialized kernel-bypass approaches. While the in-kernel approach guarantees safety and provides resource multiplexing, it imposes too much overhead on microsecond-scale tasks. Kernel-bypass approaches improve throughput substantially but sacrifice safety and complicate resource management: if applications are mutually distrusting, then either each application must have exclusive access to its own device or else the device itself must implement resource management.

This paper shows how to attain both safety and performance via intra-process isolation for data plane libraries. We propose *protected libraries* as a new OS abstraction which provides separate user-level protection domains for different services (e.g., network and in-memory database), with performance approaching that of unprotected kernel bypass. We also show how this new feature can be utilized to enable sharing of data plane libraries across distrusting applications. Our proposed solution uses Intel's memory protection keys (PKU) in a safe way to change the permissions associated with subsets of a single address space. In addition, it uses hardware watchpoints to delay asynchronous event delivery and to guarantee independent failure of applications sharing a protected library.

We show that our approach can efficiently protect high-throughput in-memory databases and user-space network stacks. Our implementation allows up to 2.3 million library entrances per second per core, outperforming both kernel-level protection and two alternative implementations that use system calls and Intel's VMFUNC switching of user-level address spaces, respectively.

1 Introduction

A principal task of an operating system (OS) is to multiplex hardware resources, making them accessible to multiple user-level applications, and to arbitrate use of those resources to satisfy system-wide performance and fairness goals. User/kernel isolation enables the OS to enforce its resource management decisions in the face of untrusted and potentially malicious applications. In recent years, however, developers have begun to move I/O management into user space for the sake of higher performance, specialization, and rapid development. This strategy is often referred to as *kernel-bypass* I/O. DPDK [21] and mTCP [24] move packet processing and transport layer processing into user space; SPDK [22] does the same for direct access to fast storage devices like Optane SSDs [19]. Accelerators like Google's TPU [25] and Nvidia's GPUs [34] also rely on kernel-bypass software stacks for low-latency hardware access and rapid evolution of drivers.

The trend toward kernel bypass has enabled significant improvements in device throughput and latency [4, 39, 40]. These gains, however, have typically come at the cost of granting an application exclusive access to a device, trusting other users of the device, or relying on the existence of a hardware-level arbitrator that virtualizes or partitions the device (e.g., SR-IOV [23]). Unfortunately, device-level resource isolation is not always available and typically lacks the flexibility to implement OS-level resource management policies.

The anticipated widespread availability of byte-addressable non-volatile memory (NVM) DIMMs [45] brings similar challenges. If NVM is mapped into a process's address space so that it can be accessed directly with application load/store instructions, a memory safety error within the process could corrupt data structures on the NVM [37]. Relying on OS kernel mechanisms e.g., a file system interface, to protect access to NVM would throw away the performance potential of direct loads and stores to persistent memory.

One can, of course, implement protection domains within an address space using a trusted compiler with static [17] or dynamic [52] checking. The static approach requires a type-safe language and is thus incompatible with many existing applications. The dynamic approach incurs overhead that is significant even in the simplest cases (e.g., when checking pointers against a single boundary address), and rises steeply for more complex address space layouts [44].

What we desire is a mechanism that allows services traditionally implemented in the kernel to be encapsulated as *protected libraries* in user space. Such a mechanism should be compatible with existing applications (i.e., via re-linking), provide fast transitions into and out of protected library routines, impose little or no cost on ordinary code, accommodate multiple protected code and data segments in a single application, and support independent failure to allow a protected library to be shared across distrusting applications. Toward that end, this paper proposes *Hodor*, a mechanism for low-overhead intra-process isolation. Hodor leverages the existence of user libraries to define protection domains for services previously offered by the kernel (e.g., file systems, network stack, device drivers, etc.). Relying on library boundaries, Hodor offers practical intra-process isolation without requiring any significant effort on the part of the application programmer. It allows multiple mutually distrusting libraries to be loaded into the same address space, providing each library (and the main application) with a different "view" of code and data, and protecting each from failures in the others. (When a failure occurs, library calls in non-erroneous protection domains are permitted to complete before the process terminates.) Hodor employs the standard function call/return interface but interposes a *trampoline* on each call to change the view of the address space to that of the library being entered.

Hodor can be used to provide instances of a protected library in multiple applications with access to shared resources. Instances of a network library, for example, might provide fast, user-level access to a NIC while enforcing rate-limiting policies that require coordination among otherwise uncoordinated and mutually distrusting applications.

We propose a concrete implementation of Hodor for recent Intel processors that is based on Intel's memory protection keys, called *Protection Keys for Userspace (PKU)* [20]. We introduce a novel method that uses hardware watchpoints (i.e., debug registers [20]) to efficiently monitor program execution and ensure the safety of our approach without relying on a trusted compiler, changes to application source code, or expensive dynamic binary translation.

We also describe two alternative implementations of Hodor's isolation: 1) using a system call to switch between page tables, and 2) using Intel's Extended Page Table (EPT) switching with VM function (VMFUNC) instructions [20]. We compare our PKU-based protection with each of the alternative solutions and demonstrate that the PKU approach offers better performance. While none of the implementations is fast enough to be used for fine-grained intra-process isolation (e.g., for shadow stacks [7] or code-pointer integrity [50]), our results show that both PKU- and VMFUNC-based approaches are able to support on the order of two million calls per second per core into a protected library.

In summary, our contributions are as follows:

- We introduce Hodor, a mechanism that provides a new OS abstraction to isolate fast data-plane libraries from both the calling application and each other.
- We propose a concrete implementation of Hodor for current Intel processors based on PKU. We present a novel method that combines binary inspection and hardware watchpoints to prevent bypassing of the PKU-based protection and safely isolate libraries linked in arbitrary x86 applications.
- We quantify the performance benefits of Hodor on real-world applications with respect to both unprotected kernel bypass and isolation based on kernel-mediated page table switching and EPT switching via VMFUNC.
- We present two proof-of-concept examples of protected libraries that share state between library instances in separate applications (with independent failure modes), and discuss challenges that must be addressed in such designs.

The following section describes in more detail the problem addressed by protected libraries, including the threats against which we protect, the assumptions we make about library code, the capabilities we provide to libraries, and the system components (signal interface, threading libraries, operating system kernel) that must be modified to ensure isolation. Sec. 3 then describes our candidate implementations. We evaluate the performance of these implementations in Sec. 4 using microbenchmarks, the Silo in-memory database [48], the DPDK data-plane library package [21], and the Redis [42] NoSQL server. Sec. 5 discusses related work. Sec. 6 summarizes our conclusions.

2 Protected Libraries

Hodor's *protected library* mechanism partitions an application into multiple domains of executable code. Each domain is granted access to some parts of the address space and denied access to other parts. Each domain has private stacks and possibly a private heap, but also shares access to some pages, allowing efficient communication with other domains. Domain transitions follow standard calling conventions, mediated by *trampoline* routines that switch to the appropriate address space view, switch stacks, set up arguments to maintain calling conventions, and possibly scrub any remaining registers to avoid information leaks. Trampolines also switch back to the caller's domain when a library call returns.

2.1 Threat Model

With Hodor, an untrusted application uses protected libraries to access protected resources. A resource might comprise ordinary memory, non-volatile memory, or a memory-mapped device. By default, an application shares its entire memory space with each protected library, but the library shares only the *trampoline* code needed for cross-domain calls. In addition, an application can optionally be modified to share only buffers with the library.

Figure 1 shows an example with user-space network and storage libraries. The storage library has default access to the

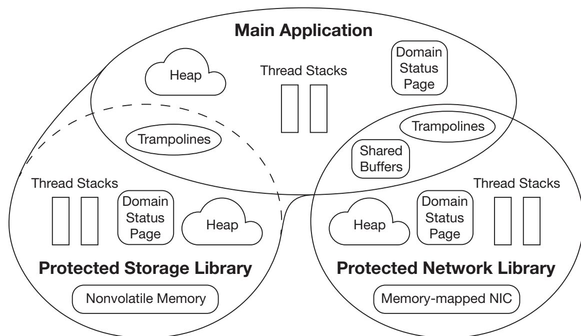


Figure 1: Protected Library Architecture. The example application has, by default, shared its entire memory space with the storage library. It has opted to share only certain buffers with the network library.

application; the network library has been given access only to shared buffers. We assume that applications may be multi-threaded and that library entry points may accept pointers to callback functions. Consequently, protected libraries must be multi-thread safe and re-entrant. When a protected library opts to share state with instances in other applications (e.g., to track resource usage and enforce fairness), the library is responsible for synchronization. We accommodate independent failures by arranging to complete the execution of any library call whose process incurs a fault in a different domain (more on this in Sec. 2.5).

As replacements for traditional kernel services, protected libraries are assumed to be written with care. Among other things, they should employ caution when dealing with potentially unsafe arguments (e.g., using methods like copy-in/copy-out) just as kernel code would. They should also ensure that transitions into other domains (e.g., invocation of a callback or a third-party library function) happen in a safe context. It would be incorrect, for example, to acquire a lock and then call an outside function, since it might terminate before returning back to the library. In our implementation, protected libraries are statically linked with all their dependencies to ensure that transitions into and out of the library conform to its API. This does not prevent use of shared libraries by the rest of the application. A more flexible implementation could dynamically link a separate instance of shared libraries in each protection domain that requires them.

A protected library is trusted to enforce security and management policies for its protected resources but is otherwise untrusted. The hardware and the operating system are part of the Trusted Computing Base (TCB), and are assumed to be correctly implemented. The application and other untrusted libraries are outside the TCB: they are not trusted to read or write protected library memory.

In this work, we are primarily interested in preserving the integrity and confidentiality of protected memory and devices from direct memory reads and writes. While side-channel attacks, and in particular those targeting transient execution [8, 27, 30], are out-of-scope, we explain the ramifications of our implementations on related transient execution attacks.

We assume that an attacker controls the application and untrusted libraries and can add arbitrary native code to the application and to untrusted libraries. We assume that the attacker cannot gain direct access to the data within a protected library's memory via the library's own API. We must consider the possibility, however, that the attacker may attempt to:

- Gain access to protected memory by changing virtual-to-physical mappings using system calls like `mmap`.
- Modify, from a compromised thread, local variables or return addresses in the stack of a thread that is running in the library.
- Subvert the loading mechanism so that a different library has access to protected memory.
- Install malicious signal handlers and then arrange for a signal to be delivered while the library is running.

We consider these issues in turn in the following subsections.

2.2 Virtual Address Space Integrity

In a standard Linux system, a process can change the page permissions of its own memory with the `mprotect` system call, and change the mappings between virtual and physical addresses with the `mmap` system call. For any given protected library *L*, we must prevent address space changes, when requested by code outside of *L*, from making *L*'s code or data accessible to the application or to another library.

This is the easiest vulnerability to address. We assume that the static and dynamic loaders are part of the trusted computing base. When asked to load a protected library, they inform the operating system of the virtual addresses used by the protected library's code and data. On any subsequent call to `mprotect`, `mmap`, etc., the kernel identifies the context in which the syscall was made (i.e., the value of `pkru` register for the PKU-based implementation and the instruction pointer for page-table switching implementations) and grants requests to change the mappings or permissions of protected library space only when made in an appropriate context.

2.3 Local Variables and Protected Stack

Within its protected memory, in addition to code, global variables, and heap, each protected library also maintains per-thread private stacks on which to store return addresses and local variables. When an application creates a new thread, we must create a new stack for each of the domains in the application. We embed this logic within the threading library (e.g., `pthreads`) so that application developers do not need to explicitly modify any application code.

When an application calls a function within a protected library, trampoline code accessible to the application must arrange for the target function to execute in the library's protection domain, where it can access its protected code and data. In particular, the trampoline must switch the stack pointer to the local stack of the calling thread. Sec. 3 describes the design of our trampolines for each intra-process isolation mech-

anism in more detail. If a protected library invokes a callback function within the application, it will also use trampoline code to switch back to the application's domain and stack.

Switching stacks can be challenging when the source or target distrusts the other. Previous work addressed this issue either by going through a trusted domain like the kernel [31] or by not supporting mutually distrusting domains [50]. While we could employ a trusted *trampoline domain*, such an implementation would double the overhead of transitions by changing the view first to the trampoline domain and then to the target domain. We address this challenge by first saving the state of the source domain (`rsp`, `fs`, etc.) in a *domain status page* accessible only in the source domain, then switching the address space view to the target domain, and finally restoring the state of the target domain from a domain status page accessible only to the target.

Like stacks, domain status pages are per-thread entities. Unfortunately, in the absence of trust, we need to access domain status pages without relying on registers such as `fs` (used for thread local storage). We can address this issue by arranging for the kernel to support a fast (vDSO-style) `gettid` call to acquire the current thread ID. The kernel maintains a list (readable, but not writable in user space) of currently running thread IDs for all CPU cores. The fast `gettid` performs a vDSO `getcpu` lookup and uses the result to find the thread ID. This enables trampoline code to access thread-local storage without relying on `fs` or performing a system call.

2.4 Program Loading

Hodor employs a trusted loader, running as root (to allow it to open I/O device files) to start up any application that uses one or more protected libraries. The trusted loader first maps all protected libraries into the virtual address space using the `mmap` system call. It then calls an initialization function within each protected library. In this function, a library can open and map the device files it needs so that it has direct read/write access to a device's memory-mapped I/O registers or to a region of persistent or shared memory. The initialization function also allocates the first stack, initializes the heap, and calls constructor functions (e.g., for C++ global variables) for the protected library.

Once all protected libraries are initialized, the trusted loader uses a system call to inform the kernel of the location of each protected library. This allows the kernel to enforce restrictions on system calls that configure the virtual address space (as per Sec. 2.2). The trusted loader then loads the application code and all other pre-loaded dynamic libraries. If inspection is required (as in the PKU-based version of Hodor that we introduce in Sec. 3.3), the loader performs it now; the kernel arranges for similar inspection on any additional libraries that are loaded on demand and on any other pages for which execute permission is enabled during execution. Finally, the trusted loader drops root privileges using the `setresuid` system call, runs the constructor functions of the application, and transfers control to the application's `main` routine.

2.5 Asynchronous Events and Termination

To support unmodified applications, Hodor must address asynchronous event delivery via signals—in particular, the possibility that the kernel might invoke a signal handler in a thread that was executing trampoline or protected library code. Such a handler might then gain access to protected library state.

In a similar vein, termination of a process while a thread is executing protected library code could leave data structures (possibly shared with library instances in other applications) in an inconsistent state or cause deadlock (by failing to release a lock). To preclude protection violations in signal handlers and to accommodate independent failures of processes whose libraries share state, we modify the kernel so that it never delivers a signal or terminates the process while threads are still running protected library or trampoline code. Instead, it places a *hardware watchpoint* (using registers `DR0`–`DR3` on an Intel machine) in the "boundary trampoline" used to exit protected libraries (line 37 of Listing 1), and delays signal delivery or termination until the watchpoint has been triggered.

As noted in Sec. 2.1, we assume that protected libraries are written with care. In particular, we assume that their operations take modest, bounded time. If a protected library does not return in a timely fashion after we install the hardware watchpoint on the trampoline (detected by expiration of a timer initiated when we arm the watchpoint), we assume that the library is defective and terminate the application (having given any other, non-defective libraries time to finish execution). We perform a similar summary cleanup if a fatal error (e.g., a `SIGSEGV`) is caused *by* library code.

Our design does not permit signal handlers to be registered for execution by a protected domain. None of the privileged library use cases we evaluated need signal handling. If they did, the kernel's signal handler API could, in principle, be extended to allow a protected library to request that a handler should execute in the library's domain. Since the kernel knows the locations of all protected library code segments in memory (see Sec. 2.4), it could confirm whether a registration request was made from trusted library code and allow or deny the request accordingly.

3 Fast Memory Isolation

This section presents three implementations of memory isolation for Hodor. The most straightforward implementation, described in Sec. 3.1, relies on separate page tables for each domain, and uses system calls to change the page table root pointer. Hodor-VMFUNC, described in Sec. 3.2, also uses per-domain page tables, and switches between them using Intel's VM Function (VMFUNC) mechanism. Both the syscall-based system and Hodor-VMFUNC rely on context identifier tags (Intel's PCID and EP4TA, respectively) to avoid flushing the translation lookaside buffer (TLB) when changing domains.

Hodor-PKU, our preferred solution (described in Sec. 3.3), uses memory protection keys to provide different access rights in the same page table. Since the access rights for each domain can be modified by user code, we need to prevent an application from bypassing our PKU-based isolation mechanism. We present a novel method in Sec. 3.3.1 that combines binary inspection and the use of hardware monitors for efficient run-time monitoring to ensure the safety of Hodor-PKU.

In an attempt to capture intuition, we speak of the domains of an application as having different "views" of a single address space. That is, conceptually, the application has a single set of virtual-to-physical mappings within which we adjust permissions on individual pages. In actuality, Hodor-VMFUNC and the syscall-based system use separate page table root pointers for separate views.

3.1 Page Table Switching via Syscalls

A straightforward way to implement protected libraries is to employ a separate page table for each domain and to use a system call to change page tables. Executable pages appear only in the tables of their corresponding domains. Protected pages rely on protection bits in the page tables of each domain to prevent undesired accesses. Unprotected application pages and trampoline pages appear in all page tables. A new system call serves to change the page table root pointer (register `CR3` on Intel machines), if and only if the requesting syscall instruction lies in the appropriate (previously registered) trampoline.

Assuming kernel page table isolation (KPTI) [13], every system call changes `CR3` on entry to the kernel. Our new syscall simply arranges (after appropriate checks) to restore the target domain's root pointer, rather than that of the calling domain, when returning to user space. This limits the overhead to only slightly more than that of a no-op system call. There may also be a rise in TLB pressure for certain applications, given that some pages will appear in the TLB more than once, with separate context tags. On hardware that has such tags, however, there is no need to flush the TLB as part of a domain switch. As a separate issue, syscalls like `munmap`, together with the TLB shootdown mechanism, are modified to remove a mapping under all applicable context tags.

In this approach, each domain of an application has a separate page table root pointer. Fortunately, the content of the tables is largely overlapping (generic heap, vDSO, kernel translations, etc.). We use a separate top-level page for each table, but many of the lower-level pages are physically shared. This approach simplifies entry manipulation and minimizes memory footprint.

3.2 Hodor-VMFUNC

Beginning with its Nehalem generation of processors, Intel has provided *extended page tables (EPT)* for virtualized environments. The traditional page table of a guest OS translates from "guest virtual" to "guest physical" addresses; the extended (second-level) page table translates from guest physical to (host) physical addresses. In the subsequent Haswell generation, Intel introduced a VM Function (VMFUNC) mechanism for fast invocation of hypervisor functions in a paravirtualized guest. This mechanism allows a guest to pre-register a set of second-level page tables and provides a (non-privileged) instruction to switch amongst them. Several systems (e.g., SeCage [32] and MemSentry [28]) have used VMFUNC to isolate a sensitive region within an application, but, they require source-code analysis and non-trivial modifications to existing applications.

Hodor-VMFUNC isolates a memory region by setting up a degenerate traditional page table that implements the identity function (with all types of access allowed) and employing a separate extended page table—analogous to the ordinary page tables of the syscall-based system—for each protection domain. An application can then switch among views with no kernel involvement. Compared to the approach of Sec. 3.1, which uses trusted kernel code to check that a domain switch is permissible, a new challenge in this approach is that we must fold the permission check into the VMFUNC instruction itself. We do so by placing the trampolines of a given library in their own page(s) and making those the only pages that are executable in the domains of both the main application and the library. A VMFUNC instruction that attempts to switch to the library's domain but lies anywhere other than an appropriate trampoline page will find the next instruction non-executable, resulting in a fault.

While the serialization overhead of an address-space-changing instruction appears inevitable (absent major architectural changes e.g., CODOMs [51] and CHERI [55], which themselves impose new overheads), using VMFUNC avoids the need for a system call when switching domains. As Sec. 4 shows, this cuts the cost of a switch by more than 50%.

Listing 1 (including the parts to the left of the vertical lines) shows the trampoline code for Hodor-VMFUNC. Line 2 saves the stack pointer of the source domain to the source domain's status page. As Sec. 2 describes, this step allows the trampoline to restore the stack when returning from the protected library; it also supports callback functions (argument copying and register scrubbing code is omitted for brevity). Line 7 sets `eax` to zero, indicating that an extended page table switch is desired. Line 8 sets `ecx` to the index (in a pre-approved table) of the domain to which to switch; line 9 effects the switch itself. Line 14 loads the stack pointer of the target domain from the target domain's status page into `rsp`; this is possible since `VMFUNC` has just enabled access to the private data of the target domain. At this point, the trampoline transfers control to a function in the target domain. Once the function returns, the trampoline saves the stack pointer of the target domain in its status page (line 19); it then resets the extended page table to the source domain (lines 25–27). Finally, it loads the source domain stack pointer from the source's status page into `rsp` (line 32) and resumes execution (line 37).

Listing 1: Hodor Trampoline: VMFUNC (left), PKU (right).

```
1  ; Save source domain stack pointer
2  movq %rsp, source_stack
3
4  ; Enable target domain view
5                              | 1:
6                              |  xorl %ecx,%ecx
7  xorl %eax, %eax             |  xorl %edx,%edx
8  movl $TGT_IDX, %ecx         |  movl $TGT_PERM, %eax
9  vmfunc                      |  wrpkru
10                             |  cmpl $TGT_PERM, %eax
11                             |  jne 1b        ; error
12                             |
13 ; Switch to target domain stack
14 movq target_stack, %rsp
15
16 ; target_domain_func()
17
18 ; Save target domain stack pointer
19 movq %rsp, target_stack
20
21 ; Disable target domain view &
22 ; Enable source domain view
23                             | 2:
24                             |  xorl %ecx,%ecx
25 xorl %eax, %eax             |  xorl %edx,%edx
26 movl $SRC_IDX, %ecx         |  movl $SRC_PERM, %eax
27 vmfunc                      |  wrpkru
28                             |  cmpl $SRC_PERM, %eax
29                             |  jne 2b        ; error
30                             |
31 ; Switch back to source domain stack
32 movq source_stack, %rsp
33 jmp BOUNDARY_TRAMP
34
35 ; Boundary Trampoline
36 BOUNDARY_TRAMP:
37 ret
```

As a starting code base, our Hodor-VMFUNC implementation uses the Dune system of Belay et al. [3], with the application running in ring 3 of VMX non-root (VM guest) mode. Running in virtualized (VMX) mode, with 2-level address translation, imposes additional overheads that are, in principle, unneeded. Most system calls, which must be handled by the operating system, incur the cost of a VM exit that is significantly more expensive than a (nonvirtualized) syscall. (That said, system calls are uncommon except during initialization in applications that use kernel-bypass data-plane libraries.) TLB refill costs increase as well, due to 2-level translation.

Ideally, we should like a hardware mechanism that allows a non-privileged instruction to switch among pre-approved page table root pointers without the need for virtualization. In the meantime, optimizations are available to mitigate the cost. First, we use huge pages to reduce the first-level (identity-function) page tables from four levels to two, eliminating half the extra cost of a VMX TLB fill. Second, it should be possible (not yet implemented) to mix virtualized and non-virtualized threads within a single application. Threads running in VMX mode will experience faster protected library calls but slower system calls; those running natively will have to use syscall-based page-table switching for library calls, but will not see additional overhead for system calls.

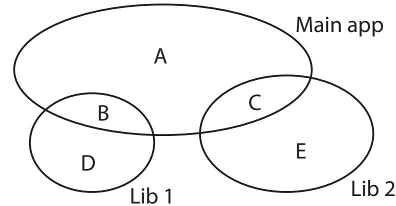


Figure 2: Address space regions in Hodor-PKU.

3.3 Hodor-PKU

In its Skylake generation of processors, Intel introduced a mechanism it calls *memory protection keys for userspace* (PKU). (Similar mechanisms have appeared in previous architectures from several other vendors.) While PKU is intended mainly as a memory safety enhancement (e.g., as a means of reducing vulnerability to stray-pointer bugs), we have realized that it can, with care, be used for protected libraries as well.

PKU [20] employs previously unused bits in each page table entry to assign a four-bit protection key to every page, allowing that page to be associated with one of 16 potential sets of access restrictions. A new 32-bit `pkru` register, writable in user space, then specifies which rights (read and/or write) should be restricted for each of the 16 key values. On every user-mode data access, the processor checks access rights in the TLB or page table as usual, then drops any rights that are found to be restricted for the PTE's key value. Since protection keys have no impact on instruction fetches (executability) and make no changes to page tables or TLB entries, the `WRPKRU` instruction, which changes the `pkru` register, does not have to serialize the pipeline, and can execute very quickly.

Hodor-PKU is based on protection keys. If we think of a protection domain as comprising a subset of the application's address space and we plot those subsets as a Venn diagram, we can assign a protection key to each separate region of the diagram and associate with each domain a `pkru` value that disables access rights for regions outside its subset of the address space. In Figure 2, the main application would disable access to regions D, and E; library 1 would disable access to regions A, C, and E; library 2 would disable access to regions A, B, and D.

Listing 1 (including the parts to the right of the vertical lines) shows the trampoline code for Hodor-PKU. Lines 6 and 7 set `ecx` and `edx` to zero; this is a required precondition of the `WRPKRU` instruction. Line 8 initializes `eax` with the appropriate set of restrictions for the domain to which the trampoline is transitioning; line 9 sets the `pkru` register to the content of `eax`. The latter change simultaneously disables the view of the source domain and enables the view of the target domain. The subsequent comparison (line 28) verifies that the expected permissions have been set, thereby avoiding an attack in which a domain puts overly generous permissions

into eax and then jumps on top of the WRPKRU instruction. Once the target function has returned and we have saved the stack pointer of the target domain (line 19), the trampoline resets the pkru register to the restrictions of the source domain (lines 24–27), and returns as in Hodor-VMFUNC.

3.3.1 Safety of Hodor-PKU

Since the processor allows user-mode code to execute the WRPKRU instruction, we must prevent a malicious application from using the instruction to attain access to a protected library's memory. One could think of employing static binary rewriting [50] to replace implicit occurrences of WRPKRU with equivalent alternatives. Unfortunately, such rewriting (including definitive determination of instruction alignment) is undecidable in the general case [41, 54], and seems inapplicable to any program that mixes read-only data into the text segment. *Dynamic* binary rewriting [6, 33] might be a viable alternative, but would likely incur prohibitive overhead (up to 2.5× for Intel Pin and 5× for DynamoRIO [33]). To address the problem, Hodor-PKU uses a trusted loader to identify all text-segment occurrences of the WRPKRU opcode outside of trampolines, and uses *hardware watchpoints* (debug registers [20]) to vet their execution at run time.

Binary Inspection The WRPKRU instruction can occur explicitly (intended by the programmer) or implicitly (unintended occurrence), as a sequence of bytes within an instruction or across the boundary between instructions. Implicit instances pose a significant threat: an adversary that seeks to bypass Hodor-PKU may attempt to corrupt control data and jump to a point in the program that happens to encode the WRPKRU instruction. By setting the contents of ecx, edx, and eax appropriately before subverting execution, the attacker could set the pkru register to any desired value, rendering the isolation useless. To address this issue, the trusted loader scans the application code and makes a list of any untrusted instances, explicit or implicit. It passes this list to the kernel, which in turn places the addresses of the potentially problematic opcodes in the debug registers. A hardware watchpoint will be triggered when any of these instructions is about to be executed, allowing the kernel to vet the instruction and let the execution proceed only if deemed safe.

Our current implementation inspects program text whenever a library is loaded and whenever execute permission for a page is enabled during execution. Once a page is marked as executable, Hodor-PKU prevents further write accesses to the page. Hodor-PKU could easily be extended to support JIT compilation by marking the faulting pages as writable but not executable, allowing JIT code to be emitted. On future attempts to execute the added code, a page fault would occur, and Hodor-PKU would reinspect the page and continue as in the current implementation.

Runtime Vetting Since the debug registers are limited in number (four—DR0 through DR3—on Intel processors [20]), we can rely on hardware to vet only a handful of WRPKRU instances at a time on each thread. Hodor therefore uses hardware watchpoints as an *LRU cache* for all the required watchpoints. Specifically, Hodor initially marks all executable pages containing WRPKRU instances as non-executable. Upon first execution, resulting in a page fault, Hodor reclaims a sufficient number of hardware watchpoints, marks the pages they formerly watched as non-executable, and uses the debug registers for the new page. If all WRPKRU instances in the page are monitored by a hardware watchpoint, Hodor marks the page as executable. In the extremely rare case of more WRPKRU instructions in a single page than the number of debug registers, we resort to single-step execution [20] for that page. We use per-thread page tables (only the root page must be unique for each thread; most lower-level pages can be shared between threads) so that the set of hardware watchpoints can be different in different threads. When watchpoints have been inserted at all appropriate locations, we rewire the page tables leading to the page containing the watchpoint for the current thread and mark it as executable.

Protection Overhead Under normal circumstances, no implicit WRPKRU will be executed. Moreover, the processor triggers a watchpoint only when a debug register points to the first byte of the executed instruction [20], so spurious traps will never occur when correctly aligned execution runs past an implicit instance.

Experiments confirm that there is no measurable overhead for this approach as long as the number of "hot" watchpoints in each thread is smaller than the number of hardware watchpoints. To assess how often this might occur, we inspected all packages in the Fedora 29 distribution for occurrences of WRPKRU. Across 58,273 rpm packages, containing about 108K executable binaries, we found only 111 binaries with a single instance of WRPKRU, 8 with two, 2 with three, none with four, and only 2 (less than 0.02%) with five or more. Most of the occurrences were implicit—typically caused by an instruction with a byte pattern ending in 0f followed by add %ebp,%edi, which has a byte pattern of 01 ef. These occurrences could easily be eliminated by modifying the compiler to insert a nop before the culprit add instructions. While such a change would not guarantee that implicit instances never occur (due to inline assembly and code generated at run time), it would almost certainly eliminate any practical performance impact.

4 Evaluation

We have evaluated Hodor using microbenchmarks and three real-world applications in which we isolated a high-throughput data-plane library or in-memory database from the rest of the application. We also constructed two proof-of-concept demonstrations of safe memory sharing among instances of a protected library in otherwise distrusting applications. We ran the microbenchmarks and in-memory database experiments on a Dell PowerEdge R640 server with two Intel Xeon Silver 4114 (Skylake) 2.20 GHz CPUs with 10 cores each and

16 GB of main memory. We ran the network experiments on Dell PowerEdge R640 servers equipped with two Intel Xeon E5-2630 v3 (Haswell) 2.40 GHz CPUs with 8 cores each and 64 GB of main memory. These machines were connected back-to-back through dual-port Mellanox ConnectX3-Pro 40 Gbps Host Channel Adapters (HCAs) to isolate their connection. All servers ran Fedora Linux 4.15 with our modifications (except for baseline experiments, which used an unmodified kernel). All machines had hyper-threading and Turbo Boost enabled.

We emulated the overhead of PKU on Haswell machines in a manner similar to previous work [28, 50]. We verified the overhead of the emulation by comparing it with the PKU transition cost on the Skylake machine.

Graphs in this section are labeled as follows:

- `unprotected`: baseline system without Hodor—kernel bypass with no intra-process isolation.
- `ptsw`: isolation via syscall-initiated page table switching, as described in Sec. 3.1.
- `ptsw-pti`: same as `ptsw`, except with kernel page-table isolation enabled.
- `vmfunc`: Hodor-VMFUNC, as described in Sec. 3.2.
- `pku`: Hodor-PKU, as described in Sec. 3.3.

Unless otherwise noted (shown in legends with `-pti`), experiments were conducted with kernel page-table isolation [13] disabled. We ran all experiments 10 times and report the arithmetic mean. We indicate 95% confidence intervals in all cases, but these are often so narrow as to be illegible in the bar graphs. The source code for Hodor is available at `http://github.com/hedayati/hodor`.

4.1 Microbenchmarks

We used microbenchmarks to measure the overhead of relevant instructions and basic operations as well as the latency of different implementations of Hodor on the Skylake machine, which supports PKU. We also implemented a no-op system call and a no-op VM call and measure their latencies. We used `rdtscp` with proper serialization [38] to measure the overhead of 1 million executions (again, computing the arithmetic mean across 10 runs).

Table 1 shows the calculated overhead of a single instance of each operation. The latency of writing to the `CR3` register impacts the syscall-based version of Hodor; the latency of `VMFUNC` and `WRPKRU` impacts Hodor-VMFUNC and Hodor-PKU, respectively. The cost of entering and leaving the kernel also impacts the syscall-based version; this cost itself depends on whether KPTI [13] is enabled. System calls with virtualization, as used in Hodor-VMFUNC, would experience the overhead of VM calls.

For reference—and to put the overheads in perspective with respect to approaches like light-weight contexts (lwC) [31] which use processes to isolate domains—we also measured the cost of a context switch caused by a semaphore and of a user-space context switch using POSIX `getcontext` and `setcontext`.

Table 1: Latency of Basic Operations

Instruction or Operation	Cycles*
write to `CR3` with `CR3_NOFLUSH`	186 ± 9
`vmfunc`	109 ± 1
`wrpkru`	26 ± 2
no-op system call w/ KPTI	433 ± 12
no-op system call w/o KPTI	96 ± 2
no-op VM call	1694 ± 131
user-space context switch	748 ± 8
process context switch using semaphore	4426 ± 41

* ± half the width of the 95% confidence interval

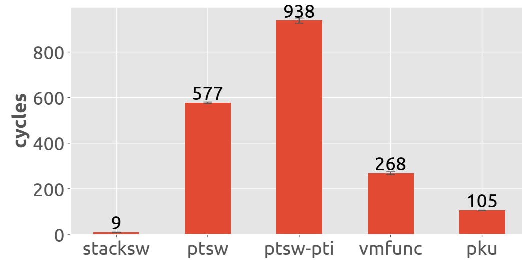


Figure 3: Transition Microbenchmarks.

Figure 3 compares the transition time from one domain to another and back again using different isolation implementations. Additionally, we measured the cost of switching stacks without providing isolation as it contributes a small amount to all implementations of Hodor. To do this, we removed the code in Listing 1 that changes domain and calls the protected library function. Figure 3 denotes the average stack switch time as `stacksw`. We also measured the cost of page table switching with kernel page-table isolation enabled; Figure 3 denotes this as `ptsw-pti`. KPTI has no impact on Hodor-VMFUNC and Hodor-PKU.

Among the implementations of isolation, Hodor-PKU has the lowest transition cost, followed by Hodor-VMFUNC. This matches the results in Table 1: changing the `pkru` register costs much less than using `vmfunc`. System calls dominate the cost of the implementations based on `ptsw`. Relative to `ptsw`, kernel page table isolation in `ptsw-pti` incurs a penalty of 62%. Stack switching itself has an almost negligible impact. As noted in Sec. 3.3, there is no measurable overhead to using debug registers to vet instances of `WRPKRU`, so long as there are no more than four watchpoints in each thread.

4.2 Silo

Silo [48] is a scalable in-memory database. It uses optimistic concurrency control and periodically-updated epochs to provide the same guarantees as a serializable database without the scalability bottlenecks. It is implemented as a library linked to the benchmark. Each benchmark thread issues transactions (of YCSB [10] or TPC-C [47]) in a loop. We configured the main Silo library as a separate domain whose pages are protected from the benchmark driver. Even in the context of a single

application, Hodor ensures that the database can be accessed only by library code—never, for example, as the result of a memory access bug in the main application. This protection may be helpful even in the course of a single execution. If the database were kept in nonvolatile memory and retained across program runs, it might be considered essential. The benchmark calls (trampolines of) library routines to perform one domain transition per transaction. All data and metadata reside in memory, and the workload is CPU intensive.

Figure 4 (i) shows the overhead of isolation for the YCSB [10] and TPC-C [47] workloads on the Skylake machine. Both use the synchronous database API in Silo, precluding batching and necessitating a very high switching rate. Both workloads were run with 20 threads.

YCSB [10] is a key-value benchmark with tiny transactions. We first filled the database with 1 million records and then ran a workload with an 80/20 read/write mix. The unmodified Silo reaches 2.27 million transactions per second on each core. Hodor incurs 44%, 54%, 27%, and 9.85% overhead in the PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations, respectively.

TPC-C [47] is a relational database benchmark with significantly larger transactions [10]. As a result, the maximum number of transactions per second is reduced to around 600,000 per core on unmodified Silo. With a lower rate of library transitions, the overhead of Hodor drops to 3%, 4.66%, 13.6%, and 1.5% for the PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations, respectively. While Hodor-VMFUNC incurs the largest overhead in this experiment, we discovered that 12% of that overhead is due to running inside a VM—apparently due to frequent use of the `nanosleep` system call in the benchmark's epoch-based garbage collector.

While we have not attempted to modify applications to remove system calls (or to replace them with equivalent functionalities that don't cause VM exits), we believe that such a change would be straightforward in this case.

4.3 DPDK TestPMD

Intel's Data Plane Development Kit (DPDK) [21] is a set of data-plane libraries that implement kernel bypass, polling drivers, and a fast packet processing framework. Packet processing applications can link against one or more of the DPDK libraries and use them to access network devices directly. We evaluate Hodor with a packet-forwarding application, `testpmd`, distributed for performance testing as a part of DPDK. Running on the Haswell machines with dual-port Mellanox ConnectX-3 HCAs, this benchmark receives raw packets from one port of the HCA and forwards them directly to another port without accessing packet contents. We connected two hosts back-to-back for endless forwarding of packets in an isolated network. We used Hodor to separate the packet-forwarding logic from the DPDK library.

Figure 4 (ii) shows the effect of Hodor on `testpmd` throughput with different thread counts and batching degrees (packets per library call / domain transition). We report throughput in packets forwarded per second as measured by `testpmd`. As a worst-case scenario for Hodor overhead, we configured the benchmark to use only a single thread and to forward packets one-by-one without batching. (Such a configuration would not be common in practice.) The unmodified DPDK in this configuration can forward more than 720,000 packets per second, and the overhead of Hodor is less than 25% with VMFUNC and 7% with PKU. As we increase the batch size (Fig. 4 (ii-a) vs. (ii-b)), the number of processed packets per transition increases and the overhead of switching becomes a smaller part of overall run time. As we provide more threads and therefore more CPUs ((b) vs. (c)), the performances of all approaches improve but the gaps decrease since the abundance of CPU resources makes the network line rate the new throughput limiter.

4.4 Redis on DPDK

Redis [42] is a NoSQL store that serves read requests from an in-memory data structure. Redis can also store data on persistent secondary storage using snapshots; we disabled this functionality in our experiments to avoid the overhead of system calls. The Redis server uses TCP to receive requests from clients. In our set-up, we use a user-space network stack called F-Stack [46] on top of the DPDK packet processing framework and driver to provide connections to Redis clients. We use Hodor to isolate the network and packet processing stack from the Redis data store logic—i.e., both F-Stack and DPDK run within the same protection domain. We run YCSB [10] on a remote client to benchmark the server configuration. Both the YCSB client and the Redis server are running on the Haswell machines, connected back-to-back via Mellanox ConnectX-3 HCAs.

The server here is the bottleneck: Redis is single-threaded; it runs a loop that waits for request arrival using an `epoll`-like call to F-Stack, receives and processes the requests, and then sends results back with F-Stack's equivalent of the `send` system call. As a result, there are at least two domain transitions per transaction.

To measure the impact of Hodor, we first loaded the Redis server with 1 million records each of length 1200 bytes. We then ran a YCSB workload [10] with a 95%/5% read-/write mix and measured how many transactions per second the Redis server supported. Figure 4 (iii) shows the results as measured and reported by the YCSB client. The unmodified server can support 220,000 transactions per second. The PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations of Hodor reduce the throughput of Redis by 12%, 35%, 5%, and 2.78%, respectively.

4.5 Discussion

The preceding subsections reveal significant performance differences among the three implementations of Hodor: syscall-based page table switching, Hodor-VMFUNC, and Hodor-

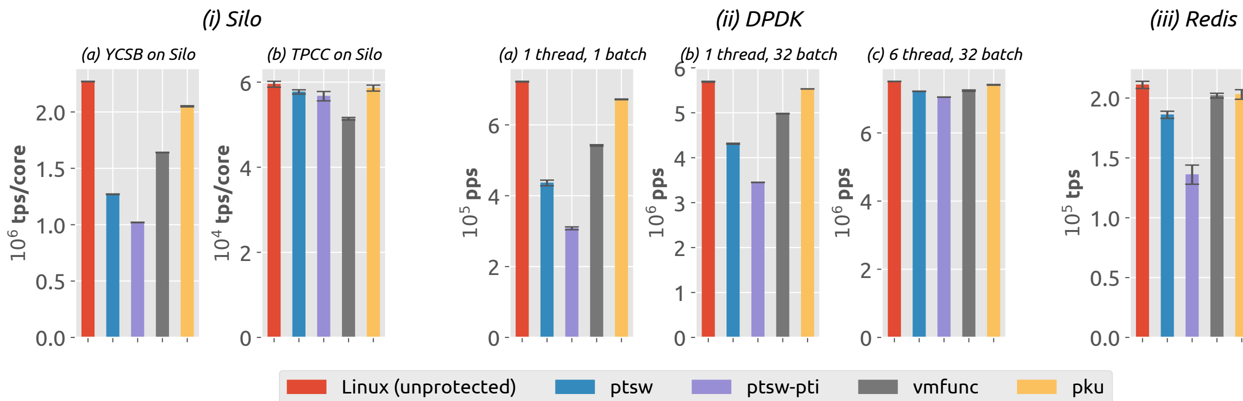


Figure 4: Hodor Overhead: (i) Silo Benchmarks, (ii) DPDK Raw Packet Forwarding Benchmarks, and (iii) Redis Benchmark.

PKU. These differences must be considered together with the issues of generality and confidentiality when applying Hodor in a particular environment.

The overhead of Hodor-PKU is very low, even at millions of domain switches per second (Fig. 4 (i-a)). With a lower number of transitions per second (Figs. 4 (i-b), 4 (iii)), possibly effected via batching or multiple worker threads (Fig. 4 (ii-b)–(ii-c)), the advantage of PKU over VMFUNC diminishes substantially. In any case, both Hodor-VMFUNC and Hodor-PKU remain considerably faster than syscall-based page table switching in most cases (Figs. 4 (ii-a)–(ii-b), 4 (iii)).

One limitation of Hodor-PKU is that current Intel hardware supports only 16 distinct memory keys. The need for a separate key for each of the regions of the "protection domain Venn diagram" (e.g., Fig. 2) thus limits us to no more than 7 mutually distrusting protected libraries in any given application—fewer if they wish to make direct calls to one another. Hodor-VMFUNC has no similar limitations on generality. There are 512 distinct function codes on current Intel machines, and a VM that uses some of these for its own purposes is still compatible with Hodor-VMFUNC. As discussed in Sec. 3.2, VMFUNC is only available in a virtualized mode. Hodor-VMFUNC, like MemSentry [28], uses Dune [3] to run applications inside a virtual machine. This restriction imposes a significant cost on system calls in the application, which now incur the latency of a VM exit; we see the impact of this latency in Figure 4 (i-b). While we don't expect frequent system calls in a data-plane library, an alternative design [32, 35] avoids VM exits on system calls by running the kernel, in addition to user programs, inside the virtual machine. Such a design has a system-wide impact on performance, since the entire software stack is virtualized, not just the intended application. Ideally, we should like to see support on future hardware for a VMFUNC-like mechanism that allows a non-privileged instruction to switch among pre-approved page table root pointers without the need for virtualization.

Absent direct access, a malicious program may attempt to steal information from protected libraries through a side channel. While such attacks are out of scope, we note that Hodor-PKU is inherently vulnerable to Meltdown-PK [8], an attack that defeats the purpose of PKU itself: the protection key bits are part of the TLB access permissions, which the processor may check late in the pipeline [8, 26]. While all Skylake processors are susceptible to this attack, the vulnerability has been fixed in more recent microarchitectures [18].

4.6 Cross-Application Sharing

Because they are protected from their calling applications, protected libraries in Hodor can, at least in principle, safely share state among library instances in separate applications. As proof-of-concept demonstrations, we have used Hodor to implement sharing in Silo and resource management in DPDK. The implementations highlight general issues that must be considered by the library designer.

Sharing in Silo: We wrote a library that uses Silo [48] internally to implement a TPCC [47]-like database. Our library provides two different views of the same database: one that can do `NewOrder`, `Payment`, and `OrderStatus` transactions, and another that can do `Delivery` and `StockLevel` transactions. Both interfaces use the same set of tables, which Silo maintains in physically shared pages. Silo guarantees consistency of the database and serializability of the transactions, while Hodor guarantees that the only way for an application to modify the database is to use the provided interface. When sharing the database between two separate applications, our library can control which interface is available to each application. More significantly, by preventing access of any kind when running outside the library, Hodor can ensure that stray memory references in a buggy application (e.g., due to out-of-bound array indexing or uninitialized pointer dereference) never compromise database invariants.

Resource management in shared DPDK: The DPDK Environment Abstraction Layer (EAL) [21] has recently added multi-process support so that mutually trusting processes can share DPDK huge-pages, memory buffers, and queues. A group of DPDK processes can then work together in a simple

transparent manner to perform packet processing or similar tasks. Using Hodor, we extended this mechanism to allow *distrusting* processes to share a single NIC. We wrote a simple library that exports several DPDK APIs (`rte_eth_rx_burst`, `rte_eth_tx_burst`, etc.). Internally, it uses shared memory to record the rate at which each application sends packets, to implement proportional share. We link this library, via Hodor, into two distrusting applications. The protected library in each application then measures its own traffic, updates shared statistics (under control of appropriate synchronization), and periodically adjusts the rate that will be allowed in the next time period.

While we were able to port the two libraries to use Hodor for sharing in just a few hundred lines of code, the experience highlighted several issues that need to be considered when tying together library instances in separate applications. As a rule of thumb, developers should think of protected libraries as extensions to the operating system. Safety-critical arguments passed by applications should be copied into library space before applying sanity checks (to avoid modification by other application threads). Libraries should also treat shared regions as potential attack vectors and should employ conventional defenses (e.g. retpolines to mitigate Spectre-type attacks [8] when relevant). Significantly, Hodor does not prevent a buggy application from invoking library routines in the "wrong" sequence, or with the "wrong" arguments. It does, however, prevent an application from undermining any invariant that is carefully maintained by those routines. It is the responsibility of the protected library to provide appropriate synchronization, scalability, and fault tolerance. The latter may be simplified by using nonblocking data structures, or by depending on Hodor to execute through to the end of library routines in the event of process failure.

5 Related Work

Hodor connects to three areas of related work: fast I/O systems that move device and resource management into user space, methods to isolate software components sharing the same virtual address space, and systems that impose security policies on operating system kernels and hypervisors.

5.1 Fast I/O Systems

Existing kernel-bypass systems do not protect libraries from untrusted applications. Arrakis [39] uses a library OS without isolation in the same address space as the application and relies on device-level SR-IOV [23] support. Device-level resource isolation policies are often rigid—e.g., limited to simple partitioning. Hodor protected libraries enable more powerful protection policies like proportional bandwidth sharing and even safe, concurrent accesses to the same data. IX [4] and ZygOS [40], both of which build on Dune [3], use virtualization to run their kernel-bypass stack in ring 0 of VMX non-root mode. While this design already isolates networking logic from the applications, it is limited to only a single trusted domain and does not support multiple distrusting data-plane libraries within the same application, as Hodor does.

Kernel-based high-throughput software stacks like MegaPipe [14] and StackMap [57] depend on aggressive batching to limit the frequency and cost of protection domain switching. Aggressive I/O batching, however, requires asynchronous programming models that are generally hard to employ and not always supported by library APIs. In Sec. 4, for example, we were unable to batch over the Silo database API [48] or F-Stack's send calls [46].

5.2 Intra-Process Isolation

There has been much previous work on intra-process isolation. The method with least overhead is to write code in a type-safe language. Work in single address space operating systems such as Singularity [17] and Verve [56] shows that application and kernel code can execute safely within the same virtual address space. The disadvantage of such systems is their incompatibility with much existing code. Hodor, in contrast, supports existing fast I/O applications.

For type-unsafe languages, approaches such as SFI [52] and XFI [49] employ either source- or binary-level instrumentation to guarantee that code cannot read or write outside of designated sections of the virtual address space. Load and store instrumentation either checks that the accessed address is within bounds or transforms out-of-bounds pointers to in-bounds pointers. SFI [52] incurs an average overhead of 17.6% for read-write protection and 4.3% overhead when only instrumenting writes. Hodor works without sophisticated binary rewriting techniques and incurs less overhead than SFI by leveraging newer hardware support.

Hardware mechanisms can isolate code running within the same virtual address space. CODOMs [51] and CHERI [55] augment instructions with capabilities. Segmentation also provides intra-process isolation [43] by requiring code to possess a *descriptor* to address a particular section of memory. By restricting which descriptors are accessible to various code components, the OS kernel can isolate untrusted components. Segmentation is supported in 32-bit but not 64-bit x86 systems [20]. ARM memory domains [2] are similar to Intel PKU [20] but available only on 32-bit processors, and memory domain permissions can be modified only in supervisor mode. Our work focuses on hardware support available in 64-bit x86 systems.

ERIM [50], developed concurrently to our work, uses protection keys like Hodor to provide an isolated domain within a single virtual address space. We believe ERIM's use of static binary rewriting to eliminate occurrences of `WRPKRU` in the application binary is insufficient to guarantee the safety of protected domains: static binary rewriting is undecidable for arbitrary x86 code [41, 54]. *Dynamic* binary rewriting (not considered in ERIM) would incur prohibitive costs, negating the performance gain of PKU.

Several OS abstractions are similar to our work. Wedge [5]

provides privilege separation and isolation among its sthreads. Each sthread is a lightweight process that inherits only a subset of the memory mappings and file descriptors of its parent, as specified in a security policy. Shreds [9] use ARM memory domains [2] to divide execution within a user-space thread. Each shred is a thread fragment with a private memory pool in which to store secret data and sensitive code. Light-weight contexts (lwCs) [31] isolate units within an address space. Each lwC has its own virtual memory mappings, file descriptors, access rights and execution state. Secure Memory Views (SMV) [15] use per-thread page tables to enforce isolation while allowing sharing between threads. SMV does not support multiple domains within a thread. Each of these systems requires a system call to change domains, while Hodor does not. Hodor can also be linked to unmodified applications.

MemSentry [28] is a memory isolation framework that provides compiler support for multiple hardware features, including EPT-switching VMFUNC and PKU, to create a safe region within a process. It analyzes and instruments applications with code which, like Hodor's trampolines, enables and disables access to the protected domain using the desired isolation mechanism. SeCage [32] uses static and dynamic compiler analysis to decompose a monolithic program into different domains and uses EPT-switching VMFUNC to prevent memory disclosure attacks even when running on a compromised OS. Unlike SeCage and MemSentry, Hodor relies on existing explicit library boundaries, alleviating the need for compiler analysis to extract components. SeCage places the entire OS and its applications in a virtual machine, while Hodor-VMFUNC and MemSentry leverage Dune's [3] process-level virtualization to expose the VMFUNC EPT-switching mechanism to individual applications. Executing the intended application in non-root mode affects the performance of that application only. SeCage [32] compensates for the system-wide performance impact of the virtualization layer with its additional protections against a malicious OS.

SkyBridge [35] uses VMFUNC to improve the latency of IPCs in a micro-kernel setting. Unlike Hodor, SkyBridge does not enforce a single way to cross protection boundaries (Hodor ensures that *only* trampolines are mapped in both source and target EPTs), which introduces the possibility for malicious VMFUNCs. To address this, SkyBridge uses static binary rewriting (inspired by ERIM [50]), which as discussed earlier, is undecidable for an arbitrary x86 binary [41, 54]. Finally, EPTI [16] uses VMFUNC to provide isolation between kernel and user-space page tables to mitigate Meltdown [8].

VMFUNC has been used for communication between components isolated at coarse granularity. High-throughput network function virtualization has used VMFUNC and EPT-switching to provide efficient communication between VMs hosting different network functions [36]. CrossOver [29] proposes a cross-world interaction mechanism that provides communication between VMs as well as different address spaces and privilege levels in or between VMs. It uses EPT-switching VMFUNCs to approximate the cost of cross-world interaction and suggests architectural changes to VMFUNC to allow such calls. While CrossOver can theoretically be used for intra-process isolation, the paper focuses on providing cross-world calls as a generic communication mechanism.

5.3 OS and Hypervisor Security

Hodor builds on previous work on security enforcement in OS kernels and hypervisors. The design of the Hodor-PKU trampoline is inspired by the Nested Kernel [12] trampoline code. Both Hodor-PKU and Nested Kernel must check that the inputs to domain switching instructions are correct because neither system enforces control flow integrity [1]. Finally, Hodor's restrictions on `mmap` to enforce code segment integrity are similar to protections in Secure Virtual Architecture [11], HyperSafe [53], and Nested Kernel [12].

6 Conclusions

We have introduced Hodor, an in-process isolation system for protection and sharing of fast data-plane libraries. Our proposed solution uses Intel's memory protection keys (PKU) to isolate components within a single address space. We also presented two alternative implementations based on separate user-level address spaces—one uses system calls for page-table switching, the other Intel's VMFUNC switching of extended page tables. Additionally, Hodor uses asynchronous event delivery and a novel application of hardware watchpoints to ensure that when multiple processes share a protected library, failure in one will not affect the others.

Our evaluation with microbenchmarks, Silo, DPDK, and Redis confirm that Hodor can provide full isolation of protected libraries while approaching unprotected kernel bypass performance. Hodor-PKU, in particular, provides 90–98% of kernel-bypass throughput in all of our experiments.

Hodor could benefit from a VMFUNC-like instruction that switches among pre-approved page table root pointers without requiring virtualization. We encourage hardware designers to consider such an extension. We would also welcome a variant of PKU with a larger number of keys and with coverage of execute rights. In future work, we hope to evaluate the cost of a Hodor implementation based on software fault isolation [52] and to explore hardware-supported implementations for additional processor architectures (e.g., ARM and Power).

Acknowledgment

We thank our shepherd, Adam Belay, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CNS-1319417, CCF-1717712, CCF-1422649, CNS-1618213 and CNS-1629770, and by a Google Faculty Research award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. on Information Systems Security*, 13:4:1–4:40, Nov. 2009.

[2] ARM Ltd. ARM Memory Domains. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html.

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Hollywood, CA, Oct. 2012.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, Oct. 2014.

[5] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 309–322, San Francisco, CA, Apr. 2008.

[6] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004. AAI0807735.

[7] N. Burow, X. Zhang, and M. Payer. Shining Light On Shadow Stacks. *arXiv e-prints*, abs/1811.03165, Nov. 2018.

[8] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv e-prints*, abs/1811.05441, Nov. 2018.

[9] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained Execution Units with Private Memory. In *37th IEEE Symp. on Security and Privacy (SP)*, pages 56–71, Oakland, CA, May 2016.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, June 2010.

[11] J. Criswell, N. Geoffray, and V. Adve. Memory Safety for Low-level Software/Hardware Interactions. In *18th USENIX Security Symp. (SEC)*, pages 83–100, Montreal, PQ, Canada, Aug. 2009.

[12] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-kernel Privilege Separation. In *20th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 191–206, Istanbul, Turkey, Mar. 2015.

[13] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In *Intl. Symp. on Engineering Secure Software and Systems (ESSoS)*, pages 161–176, Bonn, Germany, July 2017.

[14] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 135–148, Hollywood, CA, Oct. 2012.

[15] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 393–405, Vienna, Austria, Oct. 2016.

[16] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang. EPTI: Efficient defence against meltdown attack for unpatched vms. In *USENIX Annual Technical Conf. (ATC)*, pages 255–266, Boston, MA, 2018. USENIX Association.

[17] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. H. O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[18] Intel Corp. Engineering New Protections Into Hardware. http://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html.

[19] Intel Corp. Intel Optane SSD DC P4800X Series. http://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-aic.html.

[20] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2018. 325462-067US.

[21] Intel Corp. Intel DPDK: Data Plane Development Kit, 2018. http://www.dpdk.org.

[22] Intel Corp. Intel SPDK: Storage Performance Development Kit, 2018. http://www.spdk.io.

[23] Intel Corp. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology, 2018. http://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[24] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, pages 489–502, Seattle, WA, Apr. 2014.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark,

J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter Performance Analysis of a Tensor Processing Unit. In *44th Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, Toronto, ON, Canada, June 2017.

[26] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *ArXiv e-prints*, abs/1807.03757, July 2018.

[27] 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 *40th IEEE Symp. on Security and Privacy (SP)*, May 2019.

[28] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *12th ACM SIGOPS European Conf. on Computer Systems (EuroSys)*, pages 437–452, Belgrade, Serbia, Apr. 2017.

[29] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In *42nd Intl. Symp. on Computer Architecture (ISCA)*, pages 375–387, Portland, Oregon, June 2015.

[30] 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 *27th USENIX Security Symp. (SEC)*, pages 973–990, Baltimore, MD, Aug. 2018.

[31] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–64, Savannah, GA, Nov. 2016.

[32] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1607–1619, Denver, CO, Oct. 2015.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

[34] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 301–316, Salt Lake City, UT, Mar. 2014.

[35] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. SkyBridge: Fast and Secure Inter-process Communication for Microkernels. In *14th EuroSys Conf. (EuroSys)*, pages 9:1–9:15, Dresden, Germany, Mar. 2019.

[36] J. Nakajima. Xen as High-performance NFV Platform, Aug. 2018. http://events.static.linuxfound.org/sites/events/files/slides/XenAsHighPerformanceNFVPlatform.pdf.

[37] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, Sept. 2017.

[38] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, Sept. 2010.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[40] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *26th Symp. on Operating Systems Principles (SOSP)*, pages 325–341, Shanghai, China, Oct. 2017.

[41] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, Sept. 1994.

[42] Redis Labs. Redis, 2018. http://www.redis.io.

[43] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proc. of the IEEE*, 63(9):1278–1308, Sept. 1975.

[44] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In

19th USENIX Security Symp. (SEC), pages 1:1–1:11, Washington, DC, Aug. 2010.

[45] L. Spelman. Reimagining the Data Center Memory and Storage Hierarchy, May 2018. newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/.

[46] Tencent Corp. F-Stack, 2018. http://www.f-stack.org.

[47] Transaction Processing Council. TPC-C Benchmark, 2018. http://www.tpc.org/tpcc.

[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, Nov. 2013.

[49] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Seattle, WA, Nov. 2006.

[50] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *arXiv e-prints*, abs/1801.06822, Nov. 2018.

[51] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *41st Intl. Symp. on Computer Architecuture (ISCA)*, pages 469–480, Minneapolis, MN, June 2014.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.

[53] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *31st IEEE Symp. on Security and Privacy (SP)*, pages 380–395, Oakland, CA, May 2010.

[54] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating Code from Data in x86 Binaries. In *2011 European Conf. on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages III 522–536, Athens, Greece, Sept. 2011.

[55] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-system Architecture for Scalable Software Compartmentalization. In *36th IEEE Symp. on Security and Privacy (SP)*, pages 20–37, San Jose, CA, May 2015.

[56] J. Yang and C. Hawblitzel. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, ON, Canada, June 2010.

[57] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency Networking with the OS Stack and Dedicated NICs. In *USENIX Annual Technical Conf. (ATC)*, pages 43–56, Denver, CO, June 2016.

A Retargetable System-Level DBT Hypervisor

Tom Spink
University of Edinburgh

Harry Wagstaff
University of Edinburgh

Björn Franke
University of Edinburgh

Abstract

System-level Dynamic Binary Translation (DBT) provides the capability to boot an Operating System (OS) and execute programs compiled for an Instruction Set Architecture (ISA) different to that of the host machine. Due to their performance-critical nature, system-level DBT frameworks are typically hand-coded and heavily optimized, both for their guest and host architectures. While this results in good performance of the DBT system, engineering costs for supporting a new, or extending an existing architecture are high. In this paper we develop a novel, retargetable DBT hypervisor, which includes guest specific modules generated from high-level guest machine specifications. Our system simplifies retargeting of the DBT, but it also delivers performance levels in excess of existing manually created DBT solutions. We achieve this by combining offline and online optimizations, and exploiting the freedom of a Just-in-time (JIT) compiler operating in a bare-metal environment provided by a Virtual Machine (VM) hypervisor. We evaluate our DBT using both targeted micro-benchmarks as well as standard application benchmarks, and we demonstrate its ability to outperform the de-facto standard QEMU DBT system. Our system delivers an average speedup of 2.21× over QEMU across SPEC CPU2006 integer benchmarks running in a full-system Linux OS environment, compiled for the 64-bit ARMv8-A ISA and hosted on an x86-64 platform. For floating-point applications the speedup is even higher, reaching 6.49× on average. We demonstrate that our system-level DBT system significantly reduces the effort required to support a new ISA, while delivering outstanding performance.

1 Introduction

System-level DBT is a widely used technology that comes in many disguises: it powers the Android Open Source Project (AOSP) Emulator for mobile app development, provides backwards compatibility for games consoles [52], implements sandbox environments for hostile program analysis [41] and enables low-power processor implementations for popular ISAs [17]. All these applications require a complete and faithful, yet efficient implementation of a guest architecture, including privileged instructions and implementation-defined behaviors, architectural registers, virtual memory, memory-mapped I/O, and accurate exception and interrupt semantics.

The broad range of applications has driven an equally broad range of system-level DBT implementations, ranging from manually retargetable open-source solutions such as QEMU [4] to highly specialized and hardware supported approaches designed for specific platforms, e.g. Transmeta Crusoe [17]. As a de-facto industry standard QEMU supports all major platforms and ISAs, however, retargeting of QEMU to a new guest architecture requires deep knowledge of its integrated Tiny Code Generator (TCG) as it involves manual implementation of guest instruction behaviors. Consequently, retargeting is time-consuming and error-prone: e.g. the official QEMU commit logs contain more than 90 entries to bugfixes related to its ARM model alone.

In this paper we present **Captive**, our novel system-level DBT hypervisor, where users are relieved of low-level implementation effort for retargeting. Instead users provide high-level architecture specifications similar to those provided by processor vendors in their ISA manuals. In an offline stage architecture specifications are processed, before an architecture-specific module for the online run-time is generated. Captive applies aggressive optimizations: it combines the offline optimizations of the architecture model with online optimizations performed within the generated JIT compiler, thus reducing the compilation overheads while providing high code quality. Furthermore, Captive operates in a virtual bare-metal environment provided by a VM hypervisor, which enables us to fully exploit the underlying host architecture, especially its system related and privileged features not accessible to other DBT systems operating as user processes.

The envisaged use of Captive is to provide software developers with early access to new platforms, possibly hosted in a cloud environment. To facilitate this goal, ease of retargetability is as important as delivering performance levels sufficient

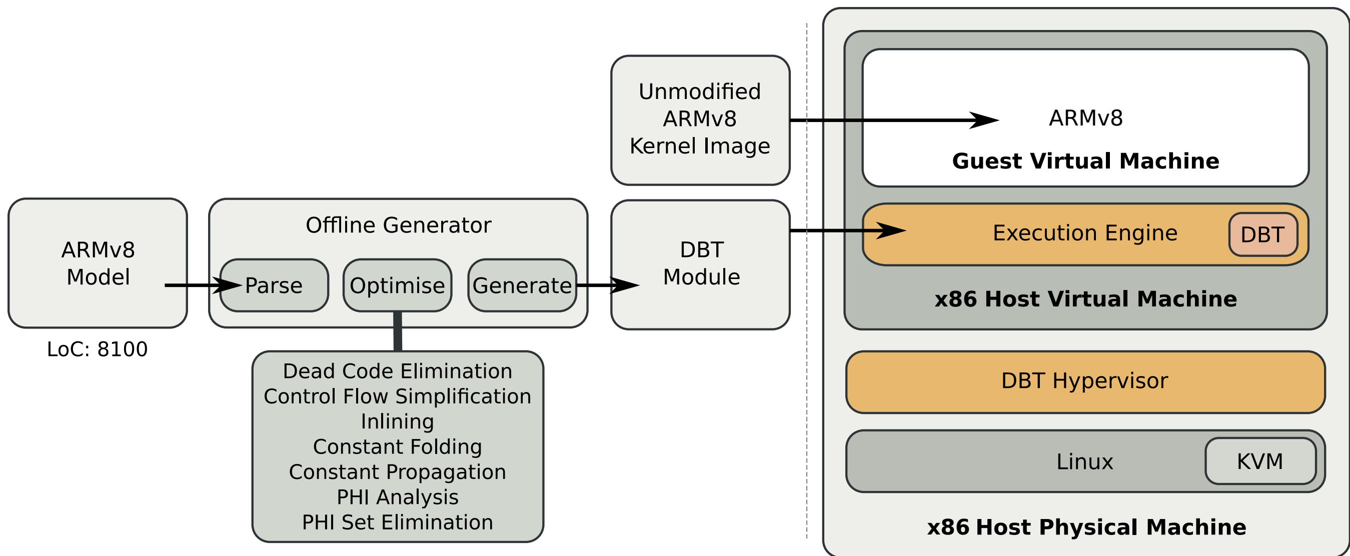


Figure 1: High-level overview of Captive.

to drive substantial workloads, i.e. software development tool chains and user applications. Whilst we currently focus on a single-core implementation, the key ideas can be translated to multi-core architectures.

We evaluate the implementation of Captive using a 64-bit ARMv8-A guest model and an x86-64 host. From a description comprising just 8100 lines of code[1] we generate a DBT hypervisor outperforming QEMU by a factor of 2.21× for SPEC CPU2006 integer applications, and up to 6.49× for floating-point workloads. This means Captive is capable of hosting a full and unmodified ARM Linux OS environment while delivering around 40% of the performance of a physical system comprising a 2.0GHz server-type Cortex-A57 processor.

1.1 Overview and Motivating Example

Figure 1 shows a high-level overview of Captive: an ARMv8-A[2] architecture description is processed by an offline tool to produce a platform-specific DBT module. Already at this stage optimizations are applied, which aid later JIT code generation. The software stack on the x86-64 host machine comprises a Kernel Virtual Machine (KVM)-based DBT hypervisor, operating on top of the host's Linux OS. This provides a *virtual* bare-metal x86-64 Host Virtual Machine (HVM) in which Captive together with the previously generated DBT module and a minimal execution engine reside to provide the Guest Virtual Machine (GVM), which can boot and run an unmodified ARMv8-A Linux kernel image. Since the JIT compiler in our system-level DBT system operates in a bare-metal HVM it has full access to the virtual host's resources and can generate code to exploit these resources.

For example, consider Figure 2. A conventional system-level DBT system hosted on an x86-64 architecture, e.g. QEMU, operates entirely as a user process in protection ring 3 on top of a host OS operating in ring 0. This means that any code generated by QEMU's JIT compiler, either guest user or system code, also operates in the host's ring 3, which restricts access to system features such as page tables. Such a system operating exclusively in ring 3 needs to provide software abstractions and protection mechanisms for guest operations, which modify guest system state. In contrast, Captive operates in VMX root mode, and provides a bare-metal HVM with rings 0-3. Our execution engine and DBT operate in the virtual machine's ring 0, and track the guest system's mode. This enables us to generate code operating in ring 0, for guest system code, and ring 3, for guest user code. This means we can use the HVM's hardware protection features to efficiently implement memory protection or allow the hypervisor to modify the HVM's page tables in order to directly map the GVM's virtual address space onto host physical memory.

Porting to a different host architecture can be accomplished by utilising similar features offered by that architecture, e.g. Arm offers virtualization extensions that are fully supported by KVM, and privilege levels `PL0` and `PL1`, which are similar to x86's ring 3 and ring 0, respectively. These similarities also enable our accelerated virtual memory system to work across platforms.

1.2 Contributions

Captive shares many concepts with existing DBT systems, but it goes beyond and introduces unique new features. We

[1] Compared to 17766 LoC for QEMU's ARM model plus a further 7948 LoC in their software floating-point implementation.

[2] Or any other guest architecture, e.g. RISC-V.

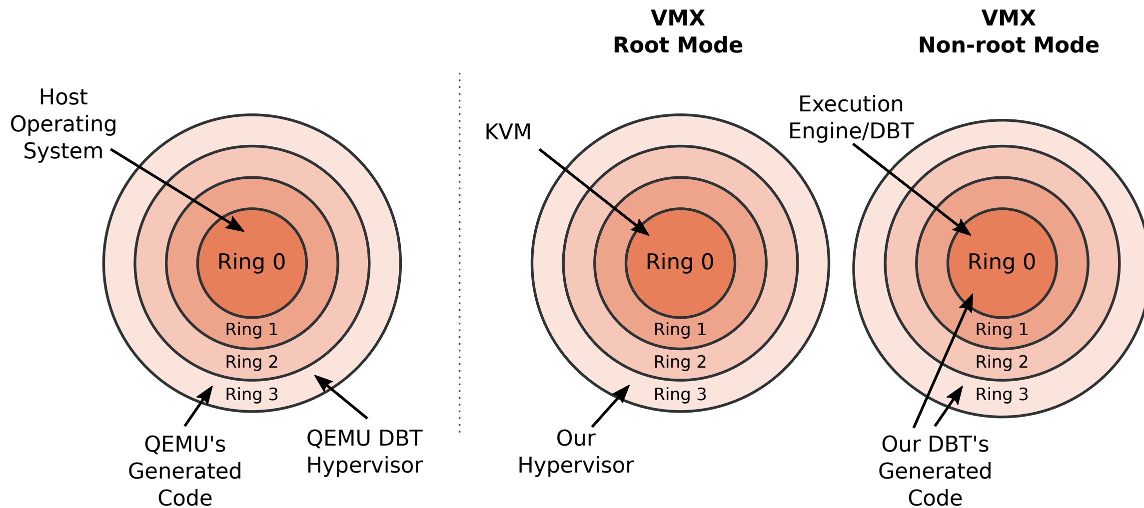


Figure 2: x86 protection rings. Ring 0 is the most privileged (kernel mode), and ring 3 is the least privileged (user mode). QEMU operates in ring 3, whereas Captive takes advantage of a host VM to operate in ring 0 and ring 3. The hypervisor component operates outside the host virtual machine, in *VMX root mode*.

	QEMU [4]	HQEMU [21]	PQEMU [18]	Walkabout [12]	Yirr-Ma [44]	ISAMAP [38]	Transmeta CMS [17]	Harmonia [31]	QuickTransit [26]	HyperMAMBO-x64 [15]	Captive (2016) [40]	MagiXen [10]	**Captive**
System-Level	✓	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓
Retargetable	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓
Architecture Description Language	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✓
Hypervisor	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
Host FP Supprt	✗	✓	✗	N/A	✗	✓	✓	✗	✓	✓	✓	✓	✓
FP bit-accurate	✓	✗	✓	N/A	N/A	N/A	✓	N/A	N/A	N/A	✗	✓	✓
Host SIMD Support	(✓)	(✓)	(✓)	✗	✗	✓	✓	✗	N/A	N/A	✗	✓	✓
64-bit support	✓	✓	✗	✓	✓	✓	✗	✗	N/A	✗	✗	✗	✓
Publicly Available	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Table 1: Feature comparison of DBT systems. Brackets indicate partial support.

provide a feature comparison in Table 1, and present further information on related work in Section 4. Among the contributions of this paper are:

1. We develop a generic system-level DBT framework, where the effort to support new guest platforms is reduced by using **high-level architecture descriptions**.

2. We use split compilation in a DBT, combining **offline and online optimization** to reduce pressure on the performance critical JIT compiler while maintaining code quality.

3. We pioneer a DBT approach where the integrated JIT compiler is part of a **DBT hypervisor** and can generate code that takes full advantage of this execution context.

Captive has been released as an open-source project, to enable community-driven development and independent performance evaluation.[3]

2 Retargetable DBT Hypervisor

2.1 Overview

In this section, we describe the key concepts of Captive, which comprises two main components, (1) an **offline** generation component, and (2) an **online** runtime component.

The offline phase involves describing the target machine architecture, and is discussed in Section 2.2. In this phase, modules for inclusion in the runtime component are generated. Complex architectural behaviour (such as the operation of the

[3] See https://gensim.org/simulators/captive

Optimization	Active in Opt. Level
Dead Code Elimination	O1–4
Unreachable Block Elimination	O1–4
Control Flow Simplification	O1–4
Block Merging	O1–4
Inlining	O1–4
Dead Variable Elimination	O1–4
Jump Threading	O2–4
Constant Folding	O3–4
Constant Propagation	O3–4
Value Propagation	O3–4
Load Coalescing	O3–4
Dead Write Elimination	O3–4
PHI Analysis	O4
PHI Elimination	O4

Table 2: Optimizations applied in the offline stage.

Memory Management Unit (MMU)) are described in regular source-code files, and compiled together with the generated source-code. The online runtime component is discussed in Section 2.3, and comprises a further two sub-components, (1) a **user-mode application** that activates and configures a KVM Virtual Machine, and (2) a **unikernel** that runs inside the KVM VM, and implements guest instruction translation and general guest machine execution.

The DBT system itself runs inside a VM with no standard OS support. Normally, a virtual machine provides a bare-metal environment in which an OS is loaded, and then user applications are executed. We instead skip the OS entirely, and implement our DBT on the virtual bare-metal hardware. Whilst this adds complexity to the implementation of the DBT, it also allows the DBT to directly use host architectural features, without having to negotiate with an OS. This is in contrast to the majority of other system-level DBTs, which typically run as user-mode applications in an OS. The trade-off here is that Captive relies on KVM, reducing host operating system portability.

2.2 Offline Stage

2.2.1 Architecture Description

The guest machine architecture is described using a high-level Architecture Description Language (ADL) that defines instruction syntax (i.e. how to decode instructions) and instruction semantics (i.e. how to execute instructions). The ADL is also used to describe architectural features, such as the register file size and layout, word sizes, endianness, etc.

The ADL is based on a modified version of ArchC [1], and our offline generator tool processes the description into an intermediate form, performs some optimization and analysis, before finally producing modules for the DBT as output.

Instruction semantics (the functional behavior of guest machine instructions) are described in a high-level C-like language. This Domain Specific Language (DSL) allows the behavior of instructions to be specified easily and naturally, by, e.g. translating the pseudo-code found in architecture manuals into corresponding C-like code.

```
1 execute(add) {
2   uint64 rn = read_register_bank(BANK0, inst.a);
3   uint64 rm = read_register_bank(BANK0, inst.b);
4   uint64 rd = rn + rm;
5   write_register_bank(BANK0, inst.a, rd);
6 }
```

Figure 3: High-level C-like representation of instruction behavior

Figure 3 provides an example description of an `add` instruction that loads the value from two guest registers (lines 3 and 4), adds them together (line 5), then stores the result to another guest register (line 6). This example shows how a typical instruction might look, and how its behavior can be naturally expressed. Of course, this is a simple example: most 'real-world' instruction descriptions contain branching paths to select specific instruction variants (e.g., flag-setting or not), more complex calculations, and floating point and vector operations, all of which can be handled by the ADL.

2.2.2 Intermediate SSA Form

During the offline phase, instruction behavior descriptions are translated into a domain-specific Static Single Assignment (SSA) form, and aggressively optimized. The optimization passes used have been selected based on common idioms in instruction descriptions. For example, very few loop-based optimizations are performed, since most individual instructions do not contain loops. Optimizing the model at the offline stage makes any simplifications utilized by the designer in the description less of a performance factor in the resulting code.

The domain-specific SSA contains operations for reading architectural registers, performing standard arithmetic operations on values of integral, floating-point and vector types, memory and peripheral device access and communication, and a variety of built-in functions for common architectural behaviors (such as flag calculations and floating point NaN/Infinity comparisons).

Additionally, meta-information about the SSA is held, indicating whether each operation is *fixed* or *dynamic*. Fixed operations are evaluated at instruction translation time, whereas dynamic operations must be executed at instruction run-time. For example, the calculation of a constant value, or control flow based on instruction fields is *fixed*, but computations which depend on register or memory values are *dynamic* [46]. Fixed operations can produce dynamic values, but dynamic operations must be executed as part of instruction emulation.

Figure 4 shows the direct translation of the instruction behavior (from Figure 3) into corresponding SSA form. A

```
action void add (Instruction sym_1_3_parameter_inst) [
  uint64 sym_14_0_rd
  uint64 sym_5_0_rn
  uint64 sym_9_0_rm
] {
  block b_0 {
    s_b_0_0 = struct sym_1_3_parameter_inst a;
    s_b_0_1 = bankregread 7 s_b_0_0;
    s_b_0_2: write sym_5_0_rn s_b_0_1;
    s_b_0_3 = struct sym_1_3_parameter_inst b;
    s_b_0_4 = bankregread 7 s_b_0_3;
    s_b_0_5: write sym_9_0_rm s_b_0_4;
    s_b_0_6 = read sym_5_0_rn;
    s_b_0_7 = read sym_9_0_rm;
    s_b_0_8 = binary + s_b_0_6 s_b_0_7;
    s_b_0_9: write sym_14_0_rd s_b_0_8;
    s_b_0_10 = struct sym_1_3_parameter_inst a;
    s_b_0_11 = read sym_14_0_rd;
    s_b_0_12: bankregwrite 0 s_b_0_10 s_b_0_11;
    s_b_0_13: return;
  }
}
```

Figure 4: Unoptimized domain-specific SSA form of the `add` instruction from Figure 3.

```
action void add (Instruction sym_1_3_parameter_inst) [] {
  block b_0 {
    s_b_0_0 = struct sym_1_3_parameter_inst a;
    s_b_0_1 = bankregread 7 s_b_0_0;
    s_b_0_2 = struct sym_1_3_parameter_inst b;
    s_b_0_3 = bankregread 7 s_b_0_2;
    s_b_0_4 = binary + s_b_0_1 s_b_0_3;
    s_b_0_5 = struct sym_1_3_parameter_inst a;
    s_b_0_6: bankregwrite 0 s_b_0_5 s_b_0_4;
    s_b_0_7: return;
  }
}
```

Figure 5: Equivalent optimized domain-specific SSA form of the `add` instruction from Figure 3.

series of optimizations (given in Table 2) are then applied to this SSA, until a fixed-point is reached. Figure 5 shows the optimized form of the SSA.

The offline optimizations allow the user to be expressive and verbose in their implementation of the model, whilst retaining a concise final representation of the user's intent. For example, dead code elimination is necessary in the case where helper functions have been inlined, and subsequently subjected to constant propagation/folding, which eliminates a particular control-flow path through the function.

2.2.3 Generator Function

The domain-specific SSA itself is not used at runtime, but instead is used in the final offline stage to build simulator-specific generator functions. These functions are either compiled in, or dynamically loaded, by the DBT, and are invoked at JIT compilation time. The generator functions call into the DBT backend, which produces host machine code. When an instruction is to be translated by the DBT, the corresponding generator function is invoked.

Figure 6 shows the corresponding generator function, produced from the optimized SSA form in Figure 5. The generator function is clearly machine generated, but *host* compiler optimizations (in the offline stage) will take care of any inefficiencies in the output source-code. Additionally (and not shown for brevity) the offline stage generates source-code comments, to assist in debugging.

2.3 Online Stage

The online stage of Captive involves the actual creation and running of the guest virtual machine. This takes the form of a KVM-based DBT hypervisor, which instantiates an empty *host virtual machine*, which then loads the *execution engine* (a small, specialized unikernel) that implements the DBT. The KVM-based portion of the hypervisor also includes software emulations of guest architectural devices (such as the interrupt controller, UARTs, etc). The DBT comprises four main phases, as shown in Figure 7: **Instruction Decoding**, **Translation**, **Register Allocation**, and finally **Instruction Encoding**.

2.3.1 Instruction Decoding

The first phase in our execution pipeline is the instruction decoder, which will decode one *guest* basic block's worth of instructions at a time. The decoder routines are automatically generated from the architecture description during the offline stage, utilizing techniques such as Krishna and Austin [27], Theiling [43].

2.3.2 Translation

During the translation phase, a generator function (that was created in the offline stage) is invoked for each decoded instruction. The generator function calls into an *invocation Directed Acyclic Graph (DAG) builder*, which builds a DAG representing the data-flow and control-flow of the instruction under translation. Operations (represented by nodes in the DAG) that have side effects result in the collapse of the DAG at that point, and the emission of low-level Intermediate Representation (IR) instructions representing the collapsed nodes.

A node with side effects is one through which control-flow cannot proceed without the state of the guest being mutated in some way. For example, a `STORE` node is considered to have side-effects, as the guest machine register file has been changed.

During emission, the tree rooted at that node is traversed, emitting IR for the operations required to produce the input values for that node. This feed-forward technique removes the need to build an entire tree then traverse it later. Collapsing nodes immediately to IR improves the performance of the DBT, as instructions are generated as soon as possible.

```
bool generator::translate_add(const test_decode_test_F1& insn, dbt_emitter& emitter) {
  basic_block *__exit_block = emitter.create_block();
  goto fixed_block_b_0;
fixed_block_b_0:  {
    auto s_b_0_1 = emitter.load_register(emitter.const_u32((uint32_t)(256 + (16 * insn.a))), dbt_types::u64);
    auto s_b_0_3 = emitter.load_register(emitter.const_u32((uint32_t)(256 + (16 * insn.b))), dbt_types::u64);
    auto s_b_0_4 = emitter.add(s_b_0_1, s_b_0_3);
    emitter.store_register(emitter.const_u32((uint32_t)(0 + (8 * insn.a))), s_b_0_4);
    goto fixed_done;
  }
fixed_done:
  emitter.jump(__exit_block);
  emitter.set_current_block(__exit_block);
  if (!insn.end_of_block) emitter.inc_pc(emitter.const_u8(4));
  return true;
}
```

Figure 6: Generator function produced from ADL code shown in Figure 3

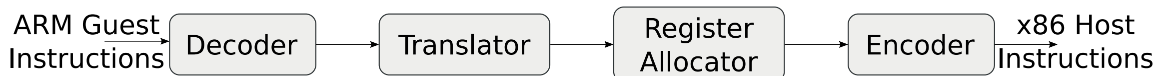


Figure 7: Online flow including decoder, translator, register allocation and instruction encoder.

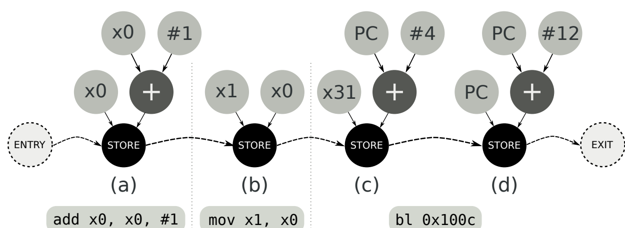


Figure 8: Example ARM assembly, and the corresponding (uncollapsed) DAG built during translation. Nodes (a), (b), (c), and (d) have side effects, causing the emission of low-level IR based on the tree rooted at that node.

```
mov (%rbp), %VREG0  ; Load guest reg. into temporary
add $1, %VREG0      ; Add one.
mov %VREG0, (%rbp)  ; Store temporary to guest reg.
mov (%rbp), %VREG1  ; Load guest reg. into temporary
mov %VREG1, 8(%rbp) ; Store temporary to guest reg.
lea 4(%r15), %VREG2 ; Load PC+4 into temporary
mov %VREG2, 0xf8(%rbp) ; Store into guest reg.
add $12, %r15       ; Increment PC by 12
```

Figure 9: As each node with side-effects is inserted into the DAG, low-level IR is emitted that implements that node. This IR represents host instructions, but with virtual registers instead of physical registers.

This strategy enables high-level operations to take place on transparent values, and implements a weak form of tree pattern matching on demand. When a node is collapsed, specializations can be made depending on how the tree is formed at the node. For example, the STORE node ((d) in Figure 8) that updates the PC by incrementing its value, can be emitted as a single x86 instruction. Instruction selection also takes place at this level, where the generator can utilize host instructions, such as fused-multiply-add when available.

In the case of an x86 host machine, the low-level IR is effectively x86 machine instructions, but with virtual register operands in place of physical registers, as shown in Figure 9. For other host machines, the IR is similar.

2.3.3 Register Allocation

After the low-level IR has been produced by the translation phase, the register allocator makes a forward pass over these instructions to discover live ranges, and then a backward pass to split live ranges into live intervals. During live-range splitting, host machine registers are allocated to virtual registers, and conflicts are resolved. Whilst not producing an optimal solution, the register allocator is fast. The allocator also marks dead instructions, so that at encoding time those instructions are ignored. Our register allocation algorithm is similar to the *simplified graph-coloring* scheme from Cai et al. [9], but with additional dead code elimination.

2.3.4 Instruction Encoding

After register allocation is complete, the low-level intermediate form of instructions can be directly lowered into machine code. The list of instructions is traversed for a final time, and the machine code is generated directly from the instruction's meta-data, into a code buffer. Any instructions that were classified as dead during register allocation are skipped.

Once machine code emission is completed, a final pass is made to apply patches to relative jump instructions, as this value is only known once each instruction has been emitted, and therefore sized.

```
1 fmov d0, #1.5    ; Store constant 1.5 in d0
2 fmul d0, d1, d2  ; Multiply d1 with d2, and store in d0
```

Figure 10: Arm floating-point input assembly

2.4 Exploiting Host Architectural Features

System-level DBT naturally involves emulating a range of guest architectural components, most notably the MMU. Traditionally, this emulation is performed in software, where each memory access must perform an *address translation* that takes a *virtual* address, and translates it via the guest page tables to a corresponding *guest physical address*. In QEMU, a cache is used to accelerate this translation, but in Captive we utilize the host MMU directly by mapping guest page table entries to equivalent host page table entries. This reduces the overhead of memory access instructions significantly, as we do not need to perform cache look-ups, and can work with the guest virtual address directly. Larger guest page sizes are supported by the host MMU directly, as multiple host pages can represent a single larger guest page. In the case of smaller guest pages, we must emulate memory accesses carefully to ensure permissions within a page are not violated. In general, we support an $n : m$ mapping between guest and host page sizes, where n, m are powers of 2.

This technique is not possible with a DBT that runs in user-mode, as the OS retains control of the host MMU page tables (although attempts have been made to emulate this by using the `mmap` system call [51]). However, with Captive, we are operating in a bare-metal environment (see Figure 1), and are able to configure the host architecture in any way we want. By tracking the protection ring of the guest machine, and executing the translated guest code in the corresponding host protection ring, we can take advantage of the host system's memory protection mechanism, for efficient implementation of guest memory protection.

We also take advantage of the x86 software interrupt mechanism (invoked using the `int` instruction), the x86 port-based I/O instructions (`in` and `out`), and the x86 fast system call instructions (`syscall` and `sysret`). These features are used to accelerate implementations of instructions that require additional non-trivial behaviors, e.g. accessing co-processors, manipulation of page tables, flushing Translation Lookaside Buffers (TLBs), and other operations specific to system-level DBT.

2.5 Floating Point/SIMD Support

In order to reduce JIT complexity, QEMU uses a software floating-point implementation, where helper methods are used to implement floating-point operations. This results in the emission of a function call as part of the instruction execution, adding significant overhead to the emulation of these instructions.

```
1  movabs  $0x3ff8000000000000, %rbp ; Store const FP value
2  mov     %rbp, 0x8c0(%r14)         ; of 1.5 in guest
3  movq    $0, 0x8c8(%r14)           ; register file.
4  lea     0x8d0(%r14), %rbp
5  mov     %rbp, %rdi
6  mov     $0x3bd, %esi
7  xor     %edx, %edx
8  callq   0x55d337b70220       ; call gvec_dup8 helper
9  lea     0x2b68(%r14), %rbp   ; Prepare arguments for
10 mov     0x9c0(%r14), %rbx    ; invocation of FP
11 mov     0xac0(%r14), %r12    ; multiply helper
12 mov     %rbx, %rdi           ; function.
13 mov     %r12, %rsi
14 mov     %rbp, %rdx
15 callq   0x55d337bd0050       ; Invoke helper
16 mov     %rax, 0x8c0(%r14)    ; Store result in
17 movq    $0, 0x8c8(%r14)      ; guest register file.
```

Figure 11: QEMU output assembly for the instruction sequence in Figure 10.

```
1 movabs $0x3ff8000000000000,%rax ; Store const FP value
2 mov    %rax,0x100(%rbp)         ; of 1.5 in guest
3 movq   $0x0,0x108(%rbp)         ; register file.
4 add    $0x4,%r15            ; Increment PC
5 movq   0x110(%rbp),%xmm0    ; Load FP multiply operand
6 mulsd  0x120(%rbp),%xmm0    ; Perform multiplication
7 movq   %xmm0,0x100(%rbp)    ; Store result
8 movq   $0x0,0x108(%rbp)
9 add    $0x4,%r15            ; Increment PC
```

Figure 12: Captive output assembly for the instruction sequence in Figure 10.

Figure 10 gives an example of two ARM floating-point instructions, which are translated by QEMU to the x86 code in Figure 11, and by Captive to the code in Figure 12. Whilst QEMU implements the `fmov` directly (lines 1—3), in much the same way as Captive, QEMU issues a function call for the floating-point multiplication (`fmul`). In contrast, Captive emits a host floating-point multiplication instruction, which operates directly on the guest register file.

Not all floating-point operations are trivial, however. Notably, there are significant differences with the way floating-point flags, NaNs, rounding modes, and infinities are handled by the underlying architecture, and in some cases this incompatibility between floating-point implementations needs to be accounted for. In these cases, Captive emits fix-up code that will ensure the guest machine state is bit-accurate with how the guest machine would normally operate. Captive only supports situations where the host machine is at least as precise as the guest. This is the most common scenario for our use cases, but in the event of a precision mismatch, we can either (a) use the x86 80-bit FPU (to access additional precision), or (b) utilise a software floating-point library.

Like QEMU, Captive emits Single Instruction Multiple Data (SIMD) instructions when translating a guest vector instruction, however QEMU's support is restricted to integer and bit-wise vector operations whereas Captive more aggressively utilizes host SIMD support.

2.6 Translated Code Management

Captive employs a code cache, similar to QEMU, which maintains the translated code sequences. The key difference is that we index our translations by *guest physical* address, while QEMU indexes by *guest virtual* address. The consequence of this is that our translations are retained and re-used for longer, whereas QEMU must invalidate all translations when the guest page tables are changed. In contrast, we only invalidate translations when self-modifying code is detected. We utilize our ability to write-protect virtual pages to efficiently detect when a guest memory write may modify translated code, and hence invalidate translations only when necessary. A further benefit is that translated code is re-used across different virtual mappings to the same physical address, e.g. when using shared libraries.

2.7 Virtual Memory Management

To accelerate virtual memory accesses in the guest, we dedicate the lower half of the host VMs virtual address space for the guest machine, and utilise the upper half for use by Captive. The lower half of the address space is mapped by taking corresponding guest page table entries, and turning them into equivalent host page table entries.

To make a memory access, the guest virtual address is masked, to keep it within the lower range, and if the address actually came from a higher address, the host page tables are switched to map the lower addresses to guest upper addresses. The memory access is then performed using the masked address directly, thus benefitting from host MMU acceleration.

3 Evaluation

Performance comparisons in the DBT space are difficult: most of the existing systems are not publicly available, and insufficient information is provided to reconstruct these systems from scratch. Furthermore, results published in the literature often make use of different guest/host architecture pairs and differ in supported features, which prohibit meaningful relative performance comparisons.[4] For this reason we evaluate Captive against the widely used QEMU DBT as a baseline, supported by targeted micro-benchmarks and comparisons to physical platforms.

3.1 Experimental Set-up

While we support a number of guest architectures, we choose to evaluate Captive using an ARMv8-A guest and an x86-64 host.[5] We conducted the following experiments on the host machine described in Table 3, and performed our comparison to native architectures on a Raspberry PI 3B, and an AMD Opteron A1100 (Table 4). We utilized both the integer and C++ floating-point benchmarks from SPEC CPU2006. Our comparisons to QEMU were made with version 2.12.1.

System	HP z440		
Architecture	x86-64	*Model*	Intel® Xeon® E5-1620 v3
Cores/Threads	4/8	*Frequency*	3.5 GHz
L1 Cache	I$128kB/D$128kB	*L2 Cache*	1MB
L3 Cache	10 MB	*Memory*	16 GB

Table 3: DBT Host System

System	AMD Opteron A1170		
Architecture	ARMv8-A	*Model*	Cortex A57
Cores/Threads	8/8	*Frequency*	2.0 GHz
L1 Cache	I$48kB/D$32kB	*L2 Cache*	4×1 MB
L3 Cache	8 MB	*Memory*	16 GB
System	Raspberry Pi 3 Model B		
Architecture	ARMv8-A	*Model*	Cortex A53
Cores/Threads	4/4	*Frequency*	1.2 GHz
L1 Cache	I$16kB/D$16kB	*L2 Cache*	512kB
L3 Cache	-/-	*Memory*	1 GB

Table 4: Native Arm Host Systems

3.2 Application Benchmarks

We have evaluated the performance of Captive and QEMU using the standard SPEC2006 benchmark suite with the *Reference* data set. As can be seen in Figure 13, we obtain significant speedups in most *Integer* benchmarks, with a geometric mean speedup of 2.2×. The two benchmarks where we experience a slow-down are `456.hmmer` and `462.libquantum`, which can be attributed to suboptimal register allocation in hot code. Figure 14 shows the speed up of Captive over QEMU on the *C++ Floating Point* portion of the benchmark suite.[6] Here we obtain a geometric mean speedup of 6.49×. This large speedup can mainly be attributed to QEMU's use of a software floating point implementation, while we use the host FPU and vector units directly.

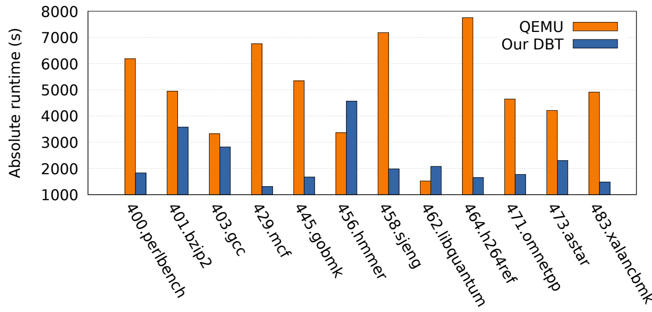
3.3 Additional Guest Architectures

We also have descriptions in our ADL for other guest architectures, detailed in Table 5. However, with the exception of ARMv7-A, these implementations currently lack full-system support. For the ARMv7-A case, we have observed similar average speed-ups of 2.5×, and up to 6× across the SPEC CPU2006 benchmark suite using Captive.

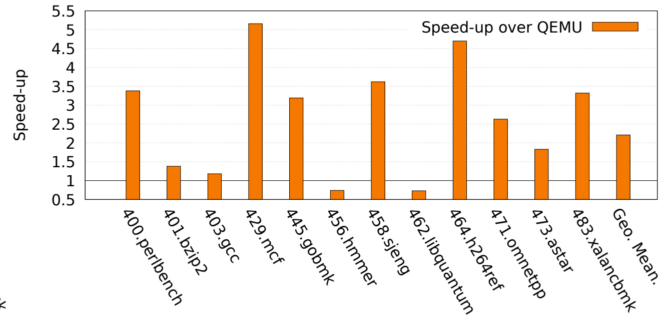
[4] For example, Harmonia [31] achieves a similar speedup of 2.2 over QEMU, but this is for user-level DBT of a 32-bit guest on a 64-bit host system whereas we achieve a speedup of 2.2 over QEMU for the harder problem of system-level DBT of a 64-bit guest onto a 64-bit host system.

[5] Additional RISC-V and x86 models will be released together with Captive.

[6] Missing Fortran benchmarks are due to the benchmarks not working both natively, and in QEMU.



(a) Absolute runtime in seconds (lower is better)



(b) Speed-up of Captive over QEMU (higher is better)

Figure 13: Application Performance: SPEC CPU2006 Integer benchmark results for Captive vs QEMU.

Architecture	Challenges	Solution
ARMv8-A	64-bit guest on 64-bit host emulation	Additional techniques for MMU emulation
ARMv7-A	If-then-else blocks, possibly spanning page boundaries	Complex control-flow handling in the JIT
x86-64	Complex instruction encoding, requiring stateful decoder.	Use of an external decoder library [23]
RISC-V	No significiant challenges	None required
TI TMS320C6x DSP	VLIW instructions, nested branch delay slots	Extensions to decoder generator, control-flow recovery
Arm Mali-G71 GPU	Complex instruction bundle headers	External "pre"-decoder for bundle headers.

Table 5: Architectures currently supported by Captive, and the architecture-specific challenges that required special attention for implementation.

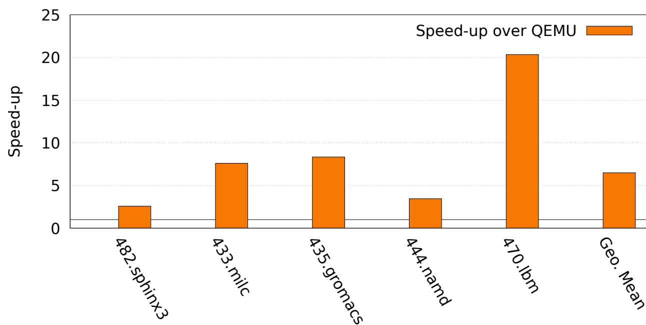


Figure 14: Speed-up of Captive over QEMU on the *Floating Point* portion of the SPEC2006 benchmark suite (higher is better)

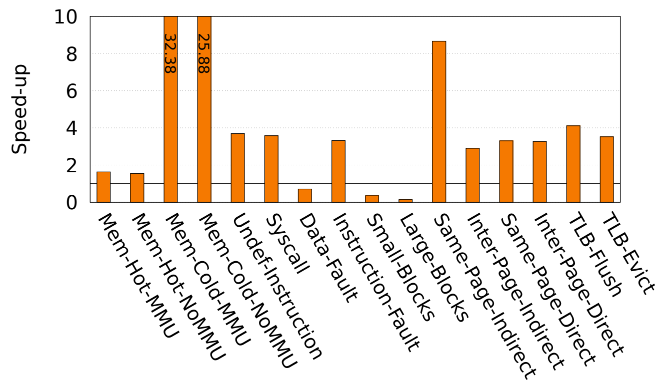


Figure 15: Speed-up of Captive over QEMU on the SimBench micro-benchmark suite

3.4 JIT Compilation Performance

Captive is on average 2.6× slower at translating guest basic blocks than QEMU. This is due in part to the more aggressive online optimizations we perform, but additionally QEMU's DBT has had years of hand-tuning, and benefits from a monolithic implementation.

However, the previous results clearly indicate that our compilation latency does not affect the runtime of the benchmarks. In fact, the extra effort we put into compilation ensures that our code quality surpasses that of QEMU's, as will be demonstrated in Section 3.6. Figure 15 shows that indeed, when using the SimBench micro-benchmark suite [47], the *Large-Blocks* and *Small-Blocks* benchmark indicate that our code generation speed is 65% and 85% slower, respectively. These benchmarks are described in Section 3.5.

Figure 16 provides a further breakdown of the time spent for JIT compilation: instruction translation (including invocation DAG generation and instruction selection) takes up more than 50% of the total JIT compilation time, followed by register allocation (including liveness analysis and dead code elimination), then host instruction encoding. Guest instruction decoding takes up 2.75% of the compilation pipeline.

We have also collected aggregate translation size statistics for `429.mcf`. We found that Captive generates larger code than QEMU, with Captive generating 67.53 bytes of host code per guest instruction, compared to QEMU's 40.26 bytes. This is due the use of vector operations in the benchmarks: while QEMU frequently emits (relatively small) function calls for

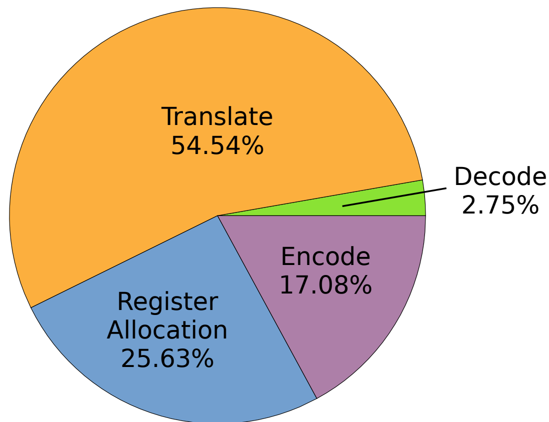


Figure 16: % time spent in each compilation phase: *Decode*, *Translate*, *Register Allocation* and *Encode*.

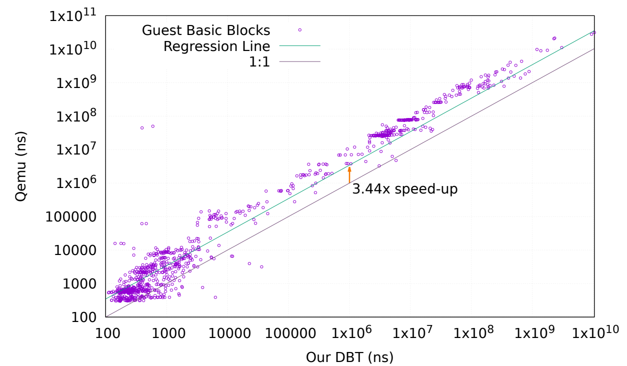


Figure 17: Measuring code quality: accumulated execution times of guest basic blocks from `429.mcf`. Blocks compiled by Captive execute, on average, 3.44× faster than their QEMU counterparts.

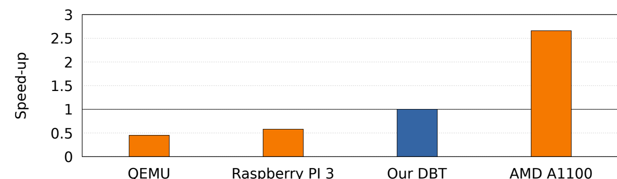


Figure 18: Comparison of Captive against native execution on two physical ARMv8-A platforms: Raspberry Pi 3 Model B & AMD Opteron A1170.

these operations, Captive emits vector operations directly. In particular, vector load and store operations require that vectors are packed and unpacked element by element, each of which can require 2–3 instructions.

3.5 Targeted Micro-Benchmarks

As well as using the SPEC benchmark suite, we have also evaluated the performance of both Captive and QEMU using SimBench[47]. This is a targeted suite of micro-benchmarks designed to analyze the performance of full system emulation platforms in a number of categories, such as the performance of the memory emulation system, control flow handling, and translation speed (in the case of DBT-based systems).

Figure 15 shows the results of running SimBench on Captive and QEMU, in terms of speedup over QEMU. Captive outperforms QEMU in most categories, except for code generation (Large-Blocks and Small-Blocks) and Data Fault handling. Captive's use of the host memory management systems results in large speedups on the memory benchmarks.

3.6 Code Quality

We assess code quality by measuring the individual basic block execution time for each block executed as part of a benchmark. For example, consider the scatter plot in Figure 17, where we show the measured aggregated block execution times across the `429.mcf` benchmark for Captive and QEMU. In order to limit the influence of infrastructure components of both platforms we have disabled block chaining for both platforms. Block execution times have been measured in the same way for both systems using the host's `rdtscp` instruction, inserted around generated native code regions representing a guest block.

A regression line and 1:1 line are also plotted in the log-log scale plot. Most points are above the 1:1 line, indicating that the vast majority of blocks are executed more quickly on Captive than on QEMU. In fact, we observe a code quality related speedup of 3.44 for this benchmark, represented by the positive shift of the regression line along the y-axis.

Further investigation reveals that Captive emits and executes, on average, 10 host instructions per guest instruction in addition to any block prologue and epilogue.

3.6.1 Impact of offline optimizations

Our offline generation system has four levels of optimization (`O1`–`O4`), although in practice we only use the maximum optimization level. These optimizations directly affect the amount of source code generated in the offline phase, where lower levels (e.g. `O1`) emit longer code sequences in the generator functions. This translates to more operations to perform at JIT compilation time, and therefore (a) larger JIT compilation latency, and (b) poorer code quality.

At the `O1` optimization level, only function inlining is performed, and results in the ARMv8A model comprising 271,299 lines of generated code. At `O4` (where a series of aggressive domain specific optimizations are performed), there is a reduction of 56%, to 120,162 lines of generated code.

	DBT System (Year)	Guest ISA	Host ISA	Distinct Contributions
User-Level DBT	Shade [13] (1993)	SPARC/MIPS	SPARC	DBT, code caching, tracing
	DAISY [19] (1997)	RS/6000	VLIW	Dyn. parallel scheduling
	FX!32 [11] (1998)	IA-32	Alpha	profiling & static BT
	UQDBT [45] (2000)	IA-32	SPARC	Retargetability
	Dynamo [2] (2000)	PA-RISC, IA-32	PA-RISC, IA-32	Same ISA Optimization
	Strata [34, 35] (2001)	SPARC/MIPS/IA-32	SPARC/MIPS	Extensibility
	Vulcan [42] (2001)	IA-32, IA-64, MSIL	IA-32, IA-64, MSIL	Het. binaries, distr. opt.
	bintrans [32] (2002)	PowerPC	Alpha	Dynamic liveness analysis
	Walkabout [12] (2002)	Retargetable (SPARC v8)	Retargetable (SPARC v9)	Arch. Descr. Lang. Interpreter and JIT generated
	DynamoRIO [7] (2003)	IA-32	IA-32	Dyn. Adapt. Optimization
	QuickTransit [26] (2004)	MIPS, PowerPC, SPARC	IA-32, IA-64, x86-64	KVM for memory translation
	Yirr-Ma [44] (2005)	Retargetable (SPARC, IA-32, ARM, PowerPC)	Retargetable (SPARC, IA-32, PowerPC)	Dyn. Opt., Part. Inlining Gen. from Spec.
	IA-32 EL [28] (2006)	IA-32	IA-64	SIMD Support
	StarDBT [48] (2007)	IA-32, x86-64	IA-32, x86-64	Trace lengthening
	N/A [6] (2008)	MIPS, VLIW	x86-64	LLVM JIT Compilation
	EHS [24] (2008)	ARC700	IA-32	Large translation regions
	Strata-ARM [30] (2009)	ARM	ARM, IA-32	Handling of exposed PC
	ISAMAP [38] (2010)	PowerPC	IA-32	Arch. Descr. Language
	ARCSim [5] (2011)	ARC700	x86-64	Parallel JIT task farm
	Harmonia [31] (2011)	ARM	IA-32	Reg. Map., Cond. codes Tiered compilation
	HQEMU [21] (2012)	ARMv7A	x86-64	Multithreaded Compilation
	HERMES [55] (2015)	IA-32, ARM	MIPS	Post-Optimization
	Pydgin [29] (2015)	ARM/MIPS	x86-64	Meta-Tracing JIT Compiler
	MAMBO-X64 [16] (2017)	AArch32	AArch64	Dyn. mapping of FP regs. Overflow address calculations Return address prediction
	HyperMAMBO-X64 [15] (2017)	AArch32	AArch64	Hypervisor support
	Pico [14] (2017)	x86-64, AArch64	x86-64, POWER8	multicore, multi-threaded DBT
System-Level DBT	Embra [53] (1996)	MIPS	MIPS	Multi-core, block chaining MMU relocation array
	Transmeta CMS [17] (2003)	IA-32	Custom VLIW	Aggressive speculation Hardware support Adaptive recompilation
	QEMU [4] (2204)	Retargetable	Retargetable	Pseudo Instructions
	MagiXen [10]	IA-32	IA-64	Integration with XEN
	PQEMU [18] (2011)	ARM	x86-64	Multi-core guest platform
	LIntel [37] (2012)	IA-32	Elbrus	Adapt. background opt.
	Captive [40]	ARMv7A	x86-64	VT Hardware Acceleration
	HybridDBT [33] (2017)	RISC-V	VLIW	Custom DBT Hardware
	Captive	Retargetable (ARMv8)	Retargetable (x86-64 + VT)	Aggressive offline optim. VM & bare-metal JIT

Table 6: Related Work: Feature comparison of existing DBT systems.

Reference	Guest ISA	Host ISA	Static/Dynamic	User/System	Distinct Contribution
Xu et al. [54]	IA-32	IA-64	Dynamic	User	Compiler Metadata
Bansal and Aiken [3]	PowerPC	IA-32	Static	User	Peephole translation rules learned by superoptimizer
Kedia and Bansal [25]	x86-64	x86-64	Dynamic	System	Kernel-level DBT
Hawkins et al. [20]	x86-64	x86-64	Dynamic	User	Optimization of Dyn. Gen. Code
Spink et al. [39]	ARMv5T	x86-64	Dynamic	User	Support for Dual-ISA
Wang et al. [49]	IA-32	x86-64	Dynamic	User	Persistent code caching
Shigenobu et al. [36]	ARMv7A	LLVM-IR	Static	User	ARM-to-LLVM IR
Wang et al. [50]	ARMv5	x86-64	Dynamic	System	Learning of translation rules
Hong et al. [22]	ARM NEON	x86 AVX2/AVX-512	Dynamic	User	Short-SIMD to Long-SIMD

Table 7: Related Work: Individual compilation techniques for Binary Translation systems.

3.6.2 Hardware Floating-point Emulation

In contrast to QEMU, Captive utilises a hardware emulated floating-point approach, where guest floating-point instructions are directly mapped to corresponding host floating-point instructions, if appropriate. Any fix-ups required to maintain bit-accuracy are performed inline, rather than calling out to helper functions. This increases the complexity of host portability, but significantly improves performance.

To determine the effect of this, we utilised a microbenchmark that exercised a small subset of (common) floating-point operations, and observed a speed-up of 2.17× of Captive (with hardware floating-point emulation) over QEMU (with software floating-point emulation). We then replaced our DBT's floating-point implementation with a software-based one (taken directly from the QEMU source-code), and observed a speed-up of 1.68×. This translates to a speed-up of 1.3× within Captive itself.

3.7 Comparison to Native Execution

We also compare the performance of Captive against two ARMv8-A hardware platforms: a Raspberry Pi 3 Model B and an AMD Opteron A1170 based server (see Table 4). The results of this comparison can be seen in Figure 18 and enable us to compare absolute performance levels in relation to physical platforms: across the entire SPEC CPU2006 suite Captive is about twice as fast as a 1.2GHz Cortex-A53 core of a Raspberry Pi 3, and achieves about 40% of the performance of a 2.0GHz Cortex-A57 core of the A1170. While outperformed by server processors it indicates that Captive can deliver performance sufficient for complex applications.

Finally, we compare the performance of Captive against native execution of the benchmarks compiled for and directly executed on the x86-64 host. Across all benchmarks we observe a speedup of 7.24 of native execution over system-level DBT, i.e. the overhead is still substantial, but Captive has significantly narrowed the performance gap between native execution, and system-level DBT.

4 Related Work

Due to their versatility DBT systems have found extensive interest in the academic community, especially since the mid-90s. In Table 6 we compare features and highlight specific contributions of many relevant DBT systems and techniques presented in the academic literature. The vast majority of existing DBT systems only provide support for user-level applications, but there also exist a number of system-level DBT approaches to which we compare Captive. In addition, numerous individual compilation techniques have been developed specifically for binary translators. Those relevant to our work on Captive are summarized in Table 7.

Captive is inspired by existing system-level DBT systems and we have adopted proven features while developing novel. Like Shade [13], Embra [53], and QEMU [4] Captive is interpreter-less and uses a basic block compiler with block chaining and trace caching. Our binary translator, however, is not hand-coded, but generated from a machine description. This allows for ease-of-retargeting comparable to Pydgin [29], but at substantially higher performance levels. Unlike Walkabout [12], Yirr-Ma [44], or ISAMAP [38], which similarly rely on machine descriptions, Captive employs split compilation and applies several optimizations offline, i.e. at module generation time, rather than relying on expensive runtime optimizations only. Instead of software emulation of floating-point (FP) arithmetic like QEMU or unsafe FP implementation like HQEMU [21], our FP implementation is bit-accurate, but still leverages the host system's FP capabilities wherever possible. Similar to IA-32 EL [28, 54] Captive translates guest SIMD instructions to host SIMD instructions wherever possible, but this mapping is generalized for any guest/host architecture pair. Like QuickTransit [26] or HyperMAMBO [15] Captive operates as a hypervisor, but provides a full-system environment rather than hosting only a single application. Captive shares this property with MagiXen [10], but provides full support for 64-bit guests on a 64-bit host rather than only 32-bit guests on a 64-bit host (which avoids address space mapping challenges introduced by same word-size system-level DBT).

5 Summary & Conclusion

In this paper we developed a novel system-level DBT hypervisor, which can be retargeted to new guest systems using a high-level ADL. We combine offline and online optimizations as well as a JIT compiler operating in a virtual bare-metal environment with full access to the virtual host processor to deliver performance exceeding that of conventional, manually optimized DBT systems operating as normal user processes. We demonstrate this using an ARMv8-A guest running a full unmodified ARM Linux environment on an x86-64 host, where Captive outperforms the popular QEMU DBT across SPEC CPU2006 application benchmarks while on average reaching 2× the performance of a 1.2GHz entry-level Cortex-A53 or 40% of a 2.0GHz server-type Cortex-A57.

5.1 Future Work

Our future work will consider support for multi- and many-core architectures, heterogeneous platforms, and support for various ISA extensions, e.g. for virtualization or secure enclaves, inside the virtualized guest system. We also plan to investigate possibilities for synthesizing guest and host architecture descriptions in the spirit of Buchwald et al. [8], or using existing formal specifications. We are also investigating a tiered compilation approach, to aggressively optimize hot code, and adding support for host retargeting, by using the same ADL as for our guest architectures.

References

[1] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5): 453–484, Oct 2005. ISSN 1573-7640. doi: 10.1007/s10766-005-7301-0. URL https://doi.org/10.1007/s10766-005-7301-0.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL http://doi.acm.org/10.1145/349299.349303.

[3] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855754.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

[5] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993508. URL http://doi.acm.org/10.1145/1993498.1993508.

[6] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *Workshop on Rapid Simulation and Performance Evalution: Methods and Tools (RAPIDO)*, 2008.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776290.

[8] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 300–313, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5617-6. doi: 10.1145/3168821. URL http://doi.acm.org/10.1145/3168821.

[9] Z. Cai, A. Liang, Z. Qi, L. Jiang, X. Li, H. Guan, and Y. Chen. Performance comparison of register allocation algorithms in dynamic binary translation. In *2009 International Conference on Knowledge and Systems Engineering*, pages 113–119, Oct 2009. doi: 10.1109/KSE.2009.16.

[10] Matthew Chapman, Daniel J. Magenheimer, and Parthasarathy Ranganathan. Magixen: Combining binary translation and virtualization. Technical Report HPL-2007-77, Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto, 2007.

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998. ISSN 0272-1732. doi: 10.1109/40.671403. URL http://dx.doi.org/10.1109/40.671403.

[12] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout: A retargetable dynamic binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[13] Robert F. Cmelik and David Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1993.

[14] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang, editors, *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 210–220. ACM, 2017. ISBN 978-1-5090-4931-8. URL http://dl.acm.org/citation.cfm?id=3049855.

[15] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, John Goodacre, and Mikel Luján. Hypermambo-x64: Using virtualization to support high-performance transparent binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 228–241, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050756. URL http://doi.acm.org/10.1145/3050748.3050756.

[16] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on arm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 333–346, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062371. URL http://doi.acm.org/10.1145/3062341.3062371.

[17] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776263.

[18] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 276–283, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4576-9. doi: 10.1109/ICPADS.2011.102. URL https://doi.org/10.1109/ICPADS.2011.102.

[19] Kemal Ebcioğlu and Erik R. Altman. Daisy: Dynamic compilation for 100 In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 26–37, New York, NY, USA, 1997. ACM. ISBN 0-89791-901-7. doi: 10.1145/264107.264126. URL http://doi.acm.org/10.1145/264107.264126.

[20] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL http://dl.acm.org/citation.cfm?id=2738600.2738610.

[21] 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*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259030. URL http://doi.acm.org/10.1145/2259016.2259030.

[22] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. Improving simd parallelism via dynamic binary translation. *ACM Trans. Embed. Comput. Syst.*, 17(3):61:1–61:27, February 2018. ISSN 1539-9087. doi: 10.1145/3173456. URL http://doi.acm.org/10.1145/3173456.

[23] Intel. Intel xed, 2018. URL https://intelxed.github.io/. Retrieved on 01/11/2018.

[24] Daniel Jones and Nigel Topham. High speed cpu simulation using ltu dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_6. URL http://dx.doi.org/10.1007/978-3-540-92990-1_6.

[25] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522718. URL http://doi.acm.org/10.1145/2517349.2522718.

[26] Paul Knowles. Transitive and QuickTransit overview, 2008.

[27] Rajeev Krishna and Todd Austin. Efficient software decoder design. *Technical Committee on Computer Architecture (TCCA) Newsletter*, October 2001.

[28] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. Optimizing dynamic binary translation for simd instructions. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.27. URL http://dx.doi.org/10.1109/CGO.2006.27.

[29] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, March 2015. doi: 10.1109/ISPASS.2015.7095811.

[30] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of dbt for the arm architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3. doi: 10.1145/1542452.1542472. URL http://doi.acm.org/10.1145/1542452.1542472.

[31] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the intel architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0698-0. doi: 10.1145/2016604.2016635. URL http://doi.acm.org/10.1145/2016604.2016635.

[32] M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 35–44, 2002. doi: 10.1109/WCRE.2002.1173062.

[33] S. Rokicki, E. Rohou, and S. Derrien. Hardware-accelerated dynamic binary translation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1062–1067, March 2017. doi: 10.23919/DATE.2017.7927147.

[34] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776265.

[35] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical report, University of Virginia, Charlottesville, VA, USA, 2001.

[36] K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota. A translation method of arm machine code to llvm-ir for binary code parallelization and optimization. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, volume 00, pages 575–579, Nov. 2018. doi: 10.1109/CANDAR.2017.75. URL doi.ieeecomputersociety.org/10.1109/CANDAR.2017.75.

[37] R. A. Sokolov and A. V. Ermolovich. Background optimization in full system binary translation. *Programming and Computer Software*, 38(3): 119–126, Jun 2012. ISSN 1608-3261. doi: 10.1134/S0361768812030073. URL https://doi.org/10.1134/S0361768812030073.

[38] Maxwell Souza, Daniel Nic$#225;cio, and Guido Araújo. Isamap: Instruction mapping driven by dynamic binary translation. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pages 117–138, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-24321-9. doi: 10.1007/978-3-642-24322-6_11. URL http://dx.doi.org/10.1007/978-3-642-24322-6_11.

[39] Tom Spink, Harry Wagstaff, Björn Franke, and Nigel P Topham. Efficient dual-isa support in a retargetable, asynchronous dynamic binary translator. In *SAMOS*, pages 103–112, 2015.

[40] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Trans. Archit. Code Optim.*, 13(4): 36:1–36:25, October 2016. ISSN 1544-3566. doi: 10.1145/2996798. URL http://doi.acm.org/10.1145/2996798.

[41] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques. *Int. J. Inf. Secur.*, 14 (2):141–153, April 2015. ISSN 1615-5262. doi: 10.1007/s10207-014-0250-0. URL http://dx.doi.org/10.1007/s10207-014-0250-0.

[42] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, April 2001. URL https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environ

[43] Henrik Theiling. Generating decision trees for decoding binaries. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, OM '01, pages 112–120, New York, NY, USA, 2001. ACM. ISBN 1-58113-426-6. doi: 10.1145/384198.384213. URL http://doi.acm.org/10.1145/384198.384213.

[44] Jens Tröger. *Specification-driven dynamic binary translation*. PhD thesis, Queensland University of Technology, 2005. URL https://eprints.qut.edu.au/16007/.

[45] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 41–51, New York, NY, USA, 2000. ACM. ISBN 1-58113-241-7. doi: 10.1145/351397.351414. URL http://doi.acm.org/10.1145/351397.351414.

[46] H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early partial evaluation in a jit-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *2013 50th ACM/EDAC/IEEE Design*

Automation Conference (DAC), pages 1–6, May 2013. doi: 10.1145/2463209.2488760.

[47] H. Wagstaff, B. Bodin, T. Spink, and B. Franke. Simbench: A portable benchmarking methodology for full-system simulators. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 217–226, April 2017. doi: 10.1109/ISPASS.2017.7975293.

[48] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 4–15, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74308-1, 978-3-540-74308-8. URL `http://dl.acm.org/citation.cfm?id=2392163.2392166`.

[49] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (dbt). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 591–603, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL `http://dl.acm.org/citation.cfm?id=3026959.3027013`.

[50] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3177160. URL `http://doi.acm.org/10.1145/3173162.3177160`.

[51] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. Hspt: Practical implementation and efficient management of embedded shadow page tables for cross-isa system virtual machines. In *ACM SIGPLAN Notices*, volume 50, pages 53–64. ACM, 2015.

[52] Tom Warren. Microsoft built an xbox 360 emulator to make games run on the xbox one, 2015. URL `https://www.theverge.com/2015/6/15/8785955/microsoft-xbox-one-xbox-360-emulator-software`.

[53] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM. ISBN 0-89791-793-6. doi: 10.1145/233013.233025. URL `http://doi.acm.org/10.1145/233013.233025`.

[54] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, and Bo Huang. Metadata driven memory optimizations in dynamic binary translator. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 148–157, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254831. URL `http://doi.acm.org/10.1145/1254810.1254831`.

[55] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. Hermes: A fast cross-isa binary translator with post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL `http://dl.acm.org/citation.cfm?id=2738600.2738631`.

MTS: Bringing Multi-Tenancy to Virtual Networking

Kashyap Thimmaraju[1] Saad Hermak[1] Gábor Rétvári[2] Stefan Schmid[3]
[1] *Technische Universität Berlin* [2] *BME HSNLab* [3] *Faculty of Computer Science, University of Vienna*

Abstract

Multi-tenant cloud computing provides great benefits in terms of resource sharing, elastic pricing, and scalability, however, it also changes the security landscape and introduces the need for strong isolation between the tenants, *also inside the network*. This paper is motivated by the observation that while multi-tenancy is widely used in cloud computing, the virtual switch designs currently used for network virtualization lack sufficient support for tenant isolation. Hence, we present, implement, and evaluate a virtual switch architecture, *MTS*, which brings secure design best-practice to the context of multi-tenant virtual networking: compartmentalization of virtual switches, least-privilege execution, complete mediation of all network communication, and reducing the trusted computing base shared between tenants. We build *MTS* from commodity components, providing an incrementally deployable and inexpensive upgrade path to cloud operators. Our extensive experiments, extending to both micro-benchmarks and cloud applications, show that, depending on the way it is deployed, *MTS* may produce 1.5-2x the throughput compared to state-of-the-art, with similar or better latency and modest resource overhead (1 extra CPU). *MTS* is available as open source software.

1 Introduction

Security landscape of cloud virtual networking. Datacenters have become a critical infrastructure of our digital society, and with the fast growth of data centric applications and AI/ML workloads, dependability requirements on cloud computing will further increase [13]. At the heart of an efficiently operating datacenter lies the idea of resource sharing and *multi-tenancy*: independent instances (e.g., applications or tenants) can utilize a given infrastructure concurrently, including the compute, storage, networking, and management resources deployed at the data center, in a physically integrated but logically *isolated* manner [38, 23].

At the level of the data center communication network, isolation is provided by the network virtualization architecture. Key to network virtualization is the *virtual switch (vswitch)*, a network component located in the Host virtualization layer of the (edge) servers that connects tenants' compute and storage resources (e.g., Virtual Machines (VMs), storage volumes, etc.), provisioned at the server, to the rest of the data center and the public Internet [38, 33, 54].

Multi-tenancy is typically provided in this design by (i) deploying the vswitches with the server's Host operating system/hypervisor (e.g, Open vSwitch aka OvS [56]); (ii) using *flow-table-level isolation*: the vswitch's flow tables are divided into per-tenant logical datapaths that are populated with sufficient flow table entries to link tenants' data-center-bound resources into a common interconnected workspace [38, 33, 54]; and (iii) overlay networks using a tunneling protocol, e.g., VXLAN [73], to connect tenants' resources into a single workspace. Alternatives to this Host-based vswitch model [56], e.g., NIC-based vswitch solutions [35, 27] and FPGA-based designs [22], share the main trait that the logical datapaths have a common networking substrate (vswitch).

Despite the wide-scale deployment [16, 22, 35], the level of (logical and performance) isolation provided by vswitches is not yet well-understood. For example, Thimmaraju et al. [69] uncovered a serious isolation problem with a popular virtual switch (OvS). An adversary could not only break out of the VM and attack all applications on the Host, but could also manifest as a worm, and compromise an entire datacenter in a few minutes. Csikor et al. [15] identified a severe performance isolation vulnerability, also in OvS, which results in a low-resource cross-tenant denial-of-service attack. Such attacks may exacerbate concerns surrounding the security and adoption of public clouds (that is already a major worry across cloud users [62]).

Indeed, a closer look at the cloud virtual networking best-practice, whereby *per-tenant logical datapaths are deployed on a single Host-based vswitch using flow-table-level isolation* [38, 33, 54], reveals that the current state-of-the-art violates basically all relevant secure system design princi-

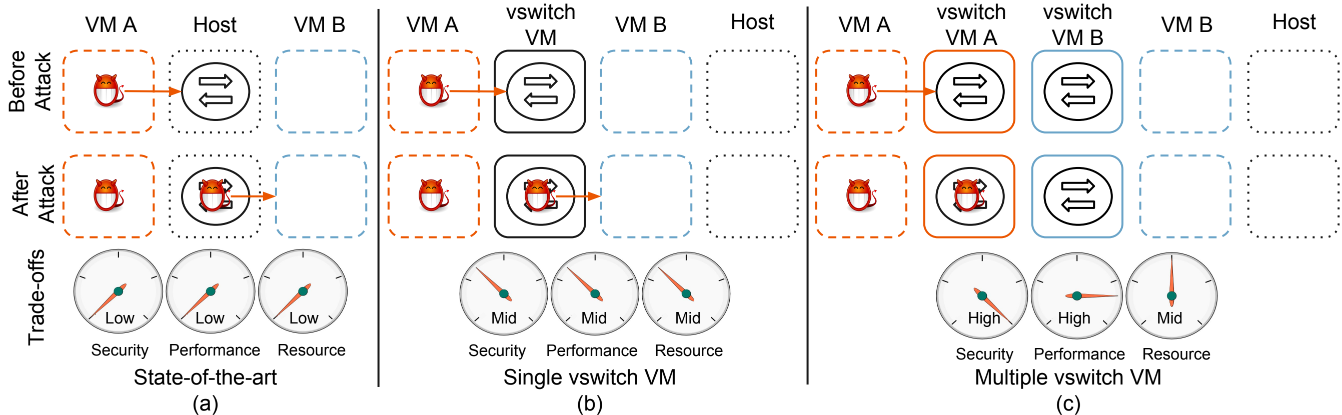


Figure 1: A high-level view of the tradeoffs between security, performance and resources for the state-of-the-art and *MTS*

ples [61, 9]. First, the principle of *least privilege* would require that any system component should be given only the minimum set of privileges necessary to complete its task, yet we see that vswitch code typically executes *on the Host* with administrator, or what is worse, with full kernel privilege [72, 28], even though this would not be absolutely necessary (see Sec. 3). Second, untrusted user code directly interacts with the vswitch and hence with the Host OS, e.g., it may send arbitrary packets from VMs, query statistics, or even install flow table entries through side channels [15], which violates the secure design principle of *complete mediation*. But most importantly, the shared vswitch design goes directly against the principle of the *least common mechanism*, which would minimize the amount of resources common to more than one tenant.

Secure vswitch design. The main motivation for our work is the observation that *current virtual switch architectures are not well-suited for multi-tenancy*. This observation leads us to revisit the fundamental design of secure vswitches. Hence, in this paper, we present, implement, and evaluate a multi-tenant (virtual) switch architecture, *MTS*, which extends the benefits of multi-tenancy to the vswitch in a secure manner, without imposing prohibitive resource requirements or jeopardizing performance.

Fig. 1 illustrates the key idea underlying the *MTS* design, by showing the security-performance-resource trade-offs for different architectures. The current vswitch architecture is shown in Fig. 1(a), whereby per-tenant logical datapaths share a common (physical or software) switch component deployed at the Host hypervisor layer (in the rest of this paper, we shall sometimes refer to this design point as the "Baseline"). As we argued above, this design is fundamentally insecure [69, 15] as it violates basic secure design principles, like least privilege, complete mediation, or the least common mechanism. In *MTS*, we address the least privilege principle by the *compartmentalization* of the vswitches (Fig. 1(b)): by moving the vswitches into a dedicated vswitch VM, we can prevent an attacker from compromising the Host via the vswitch [34]. Then, we establish a secure communication channel between the tenant VMs and the vswitch VM via a trusted hardware technique, Single Root Input/Output Virtualization, or SR-IOV, a common feature implemented in most modern NICs and motherboards [4, 35]. Thus, all tenant-to-tenant and tenant-Host networking is *completely mediated* via the SR-IOV NIC. Adopting Google's *extra security layer* design principle [1] which requires that between any untrusted and trusted component, there have to be at least two distinct security boundaries [7, 23], we introduce a second level of isolation by moving the vswitch, deployed into the vswitch VM, to the user space. Hence, at least two independent security mechanisms need to fail (user-kernel separation and VM-isolation) for the untrusted tenant code to gain access to the Host.

Interestingly, we are able to show the resultant secure vswitch design, which we call the *single vswitch VM* design, does not come at the cost of performance; just the contrary, our evaluations show that we can considerably improve throughput and latency, for a relatively small price in resources. Finally, we introduce a "hardened" *MTS* design that we call the *multiple vswitch VMs* design (Fig. 1(c)), whereby, in line with the principle of the least common mechanism, we further separate the vswitch by creating multiple separate vswitch VMs, one for each tenant (or based on security zones/classes). This way, we can maintain full network isolation for multiple tenants.

Contributions. Our main contributions in this paper are:

- We *identify requirements and design principles* that can prevent the virtual switch from being a liability to virtualization in the cloud, and we carefully apply these principles to revisit multi-tenancy in virtual networking.

- We *present* MTS*, a secure vswitch design* whereby the vswitch is moved into a separate VM that prevents malicious tenants from compromising the Host via the virtual switch, and we also show a "hardened" *MTS* design that also prevents compromising *other* tenants' virtual

networks through the vswitch. All our designs are incrementally deployable, providing an inexpensive deployment experience for cloud operators.

- We *report on extensive experiments* with our *MTS* prototype and we find a noteworthy improvement (1.5-2x) in throughput compared to the Baseline, with similar or better latency for an extra CPU. We build our prototype from off-the-shelf commodity components and existing software; *MTS* and the data from this paper are available at:

  `https://www.github.com/securedataplane`

Organization. We dive deeper into designing a secure vswitch in Section 2. In Section 3 we elaborate on *MTS* and report on two evaluations in Sections 4 and 5. We enter a discussion of *MTS* in Section 6, review related work in Section 7 and finally draw conclusions in Section 8.

2 Securing Virtual Switches

As demonstrated in previous work [15, 69], the current state-of-the-art in virtual switch design can be exploited to not only break network isolation, but also to break out of a virtual machine. This motivates us to identify requirements and design principles that make virtual switches a dependable component of the data center [68].

2.1 State-of-the-Art

Virtual networks in cloud systems using virtual switches typically follow a *monolithic* architecture, where a single controller programs a *single vswitch* running in the Host OS with per-tenant logical datapaths in the vswitch. Isolation between tenants is at the level of flow-tables [38, 33, 54]: the controller populates the flow tables in each per-tenant logical datapath with sufficient flow rules to connect the tenant's Host-based VMs to the rest of the data center and the public Internet. Those sets of flow rules are complex: with a small error in one rule potentially having security consequences, e.g., making intra-tenant traffic visible to other tenants.

As shown in Table 1, nearly all vswitches are *monolithic* in nature. A single vswitch is installed with flow rules for all the tenants hosted on the respective server. This increases the trusted computing base (TCB) of the single vswitch, as it is responsible for Layer 2-7 of the virtual networking stack. Next, nearly 80% of the surveyed vswitches are co-located with the Host virtualization layer. This increases the TCB of the server since a vswitch is a complex piece of software, consisting of tens of thousands of lines of code. The complexity of network virtualization is further increased by the fact that packet processing for roughly 70% of the virtual switches is spread across user space and the kernel (see last two columns in Table 1). These concerns are partially

Table 1: Design characteristics of virtual switches.

Name	Ref.	Year	Emphasis	Monolithic	Co-Location	Kernel	User
OvS	[55]	2009	Flexibility	✓	✓	✓	✓
Cisco NexusV	[71]	2009	Flexibility	✓	✓	✓	✗
VMware vSwitch	[72]	2009	Centralized control	✓	✓	✓	✗
Vale	[59]	2012	Performance	✓	✓	✓	✗
Research prototype	[34]	2012	Isolation	✓	✗	✓	✓
Hyper-Switch	[57]	2013	Performance	✓	✓	✓	✓
MS HyperV-Switch	[44]	2013	Centralized control	✓	✓	✓	✗
NetVM	[29]	2014	Performance, NFV	✓	✓	✗	✓
sv3	[65]	2014	Security	✗	✓	✗	✓
fd.io	[67]	2015	Performance	✓	✓	✗	✓
mSwitch	[28]	2015	Performance	✓	✓	✓	✗
BESS	[8]	2015	Programmability, NFV	✓	✓	✗	✓
PISCES	[63]	2016	Programmability	✓	✓	✓	✓
OvS with DPDK	[60]	2016	Performance	✓	✓	✗	✓
ESwitch	[46]	2016	Performance	✓	✓	✗	✓
MS VFP	[21]	2017	Performance, flexibility	✓	✓	✓	✗
Mellanox BlueField	[42]	2017	CPU offload	✓	✗	✓	✓
Liquid IO	[52]	2017	CPU offload	✓	✗	✓	✓
Stingray	[27]	2017	CPU offload	✓	✗	✓	✓
GPU-based OvS	[70]	2017	Acceleration	✓	✓	✓	✓
MS AccelNet	[22]	2018	Performance, flexibility	✓	✓	✓	✗
Google Andromeda	[16]	2018	Flexibility and performance	✓	✓	✗	✓

addressed by the current industry trend towards offloading vswitches to *smart NICs* [35, 52, 27, 42]. Indeed consolidating the vswitch into the NIC can improve the security as it reduces the TCB of the Host. These burgeoning architectures, however, share the main trait that the per-tenant logical datapaths are monolithic, often with full privilege and direct access to the Host OS, which when compromised can break network isolation and be used as a stepping-stone to the Host.

2.2 Threat Model

We assume the attacker's goal is to either escape network virtualization by compromising the virtual switch, or to tamper with other tenant's network traffic by controlling the virtual switch [69]. Hence, she can affordably rent a VM in an Infrastructure-as-a-Service (IaaS) cloud, or has somehow managed to compromise a VM, e.g., by exploiting a webserver vulnerability [14]. From the VM she can send arbitrary packets, make arbitrary computations, and store arbitrary data. However, she does not have direct control to configure the Host OS and hardware: all configuration access happens through a dedicated cloud management system.

The defender is a public cloud provider who wants to prevent the attacker from compromising virtual network isolation; in particular, *the cloud provider wants to maintain tenant-isolation even when the vswitch is compromised*. We assume that the cloud provider already supports SR-IOV at NICs [4, 35, 22] and the underlying virtualization and net-

work infrastructure is trusted, including the hypervisor layer, NICs, firmware, drivers, core switches, and so on.

2.3 Design Principles and Security Levels

Our *MTS* design is based on the application of the secure system design principles, established by Saltzer et al. [61] (see also Bishop [9] and Colp et al. [12]), to the problem space of virtual switches.

Least privilege vswitch. The vswitch should have the minimal privileges sufficient to complete its task, which is to process packets to and from the tenant VMs. Doing so limits the damage that can result from a system compromise or misconfiguration. Current best-practice is, however, to run the vswitch co-located with the Host OS and with elevated privileges; prior work has shown the types and severity of attacks that can happen when this principle fails [69]. A well-known means to the principle of least privilege is *compartmentalization*: execute the vswitch in an isolated environment with limited privileges and minimal access to the rest of the system. In the next section, we will show how *MTS* implements compartmentalization by committing the vswitches into one or more dedicated vswitch VMs.

Complete mediation of tenant-to-tenant and tenant-to-host networking. This principle requires that the network communication between the untrusted tenants and the trusted Host is completely mediated by a trusted intermediary to prevent undesired communication. This principle, when systematically applied, may go a long way towards reducing the vswitch attack surface. By *channeling all network communication* between untrusted and trusted components *via a trusted intermediary* (a so called reference monitor), the communication can be validated, monitored and logged based on security policies. In the next section, we show how complete mediation is realized in *MTS* using a secure SR-IOV channel between the tenant VMs, vswitches and Host.

Extra security boundary between the tenant and the host. This security principle, widely deployed at Google [1], requires that between any untrusted and trusted component there has to be at least two distinct security boundaries, so at least two independent security mechanisms need to fail for the untrusted component to gain access to the sensitive component [7]. We establish this extra layer of security in *MTS* by *moving the vswitch to user space*. This also contributes to implementing the "least privilege" principle: the user-space vswitch can drop administrator privileges after initialization.

Least common mechanisms. This principle addresses the amount of infrastructure shared between tenants; applied to the context of vswitches this principle requires that the network resources (code paths, configuration, caches) common to more than one tenant should be minimized. Indeed, every shared resource may become a covert channel [9]. Correspondingly, *decomposing the vswitches themselves into multiple compartments* could lead to hardened vswitch designs.

Security levels. From these principles, we can obtain different levels of security:

- **Baseline:** The per-tenant logical datapaths are consolidated into a single physical or software vswitch that is co-located with the Host OS.
- **Level-1:** Placing the vswitch in a dedicated compartment provides a first level of security by protecting from malicious tenants to compromise the Host OS via the vswitch ("single vswitch VM" in Fig. 1b).
- **Level-2:** Splitting the vswitches into multiple compartments (based on security zones or on a per-tenant basis) adds another level of security, by isolating tenants' vswitches from each other ("multiple vswitch VMs" in Fig. 1c).
- **Level-3:** Moving the vswitches into user space, combined with Baseline or Level-1 or -2, reduces the impact of a compromise and further reduces the attack surface.

3 The *MTS* Architecture

We designed *MTS* with secure design principles from Section 2.3. We first provide an overview and then present our architecture in detail.

3.1 Overview

Compartmentalization. There are many ways in which isolated vswitch compartments can be implemented: full-blown VMs, OS-level sandboxes (jails, zones, containers, plain-old user-space processes [23], and exotic combinations of these [36, 26]), hardware-supported enclaves (Intel's SGX) [50, 3], or even safe programming language compilers (Rust), runtimes (eBPF), and instruction sets (Intel MPX). For flexibility, simplicity, and ease of deployment, *MTS relies on conventional VMs as the main unit of compartmentalization*.

VMs provide a rather strong level of isolation and are widely supported in hardware, software, and management systems. This in no way means that VM-based vswitches are mandatory for *MTS*, just that this approach offers the highest flexibility for prototyping. For simplicity, Fig. 2 depicts two vswitch compartments (Red and Blue solid boxes) running independent vswitches in their isolated VMs. The multiple compartments further reduce the common mechanisms between the vswitch and the connected tenants, achieving security Level-2. Security Level-1, although not depicted, would involve only a single vswitch VM.

Complete mediation. To mediate all interactions between untrusted tenant code and the Host OS through the vswitch, we need a secure and high-performance communication medium between the corresponding compartments/VMs. In

MTS we use Single Root IO Virtualization (SR-IOV) to interconnect the vswitch compartments (see Figure 2).

SR-IOV is a PCI-SIG standard to make a single PCIe device, e.g., a NIC, appear as multiple PCIe devices that can then be attached to multiple VMs. An SR-IOV device has one or more physical functions (PFs) and one or more virtual functions (VFs), where the PFs are typically attached to the Host and the VFs to the VMs. Only the Host OS driver has privileges to configure the PFs and VFs. The NIC driver in the VMs in turn have restricted access to VF configuration. Only via the Host, VFs and PFs can be configured with unique MAC addresses and Vlan tags. Network communication between the PFs and VFs occurs via an L2 switch implemented in the NIC based on the IEEE Virtual Ethernet Bridging standard [37]. This enables Ethernet communication not only from and to the respective VMs (vswitch and tenants) based on the destination VF's MAC address but also to the external networks.

Sharing the NIC SR-IOV VF driver and the Layer 2 network virtualization mechanism implemented by the SR-IOV NIC is considerably simpler than including the NIC driver and the entire network virtualization stack (Layer 2-7) in the TCB. Tenants already share SR-IOV NIC drivers in public clouds [4, 35, 5]. Virtual networks can be built as we will see next, as per-tenant user-space applications implementing Layer 3-7 of the virtual networking stack.

Thanks to the use of SR-IOV in *MTS*, packets to and from tenant VMs completely bypass the Host OS; instead, all potentially malicious traffic is channeled through the trusted hardware medium (SR-IOV NIC) to the vswitch VM(s). Furthermore, using SR-IOV reduces CPU overhead and improves performance (see Section 4). Finally, SR-IOV provides an attractive upgrade path towards fully offloaded, smart-NIC based virtual networking: chip [39] and OS vendors [74, 66] have been supporting SR-IOV for many years now at a reasonable price, major cloud providers already have SR-IOV NICs deployed in their data centers [4, 35, 5], and, perhaps most importantly, this design choice liberates us from having to re-implement low-level and complex network components [34]: we can simply use any desired vswitch, deploy it into a vswitch VM, configure and attach VFs to route tenants' traffic through the vswitch, and start processing packets right away.

User-space packet processing. As discussed previously, we may choose to deploy the vswitches into the vswitch VM user-space to establish an extra security boundary between the tenant and the Host OS (Level-3 design). Thanks to the advances in kernel bypass techniques, several high-performance and feature-rich user-space packet processing frameworks are available today, such as Netmap [58], FD.IO [67], or Intel's DPDK [30]. Our current design of *MTS leverages OvS with the DPDK datapath for implementing the vswitches* [60]. DPDK is widely supported, it has already been integrated with popular virtual switch products,

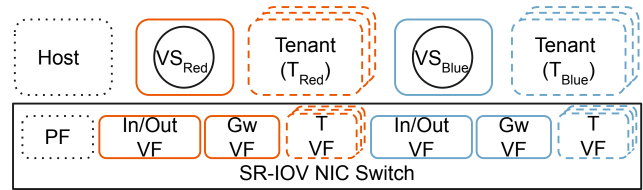


Figure 2: High-level overview of *MTS* in security Level-2. The Red and Blue vswitch compartments (VMs) are allocated dedicated virtual functions (VFs) to communicate with external networks using the *In/Out VF*, their respective tenants using the *Gw VF* and *T VF*. Communication between the vswitches, tenants and the Host physical function (PF) are mediated via the SR-IOV NIC switch.

and extensive operational experience is available regarding the expected performance and resource footprint [40]. Note, however, that using DPDK and OvS is not mandatory in *MTS*; in fact, thanks to the flexibility provided by our VM-compartments and SR-IOV, we can deploy essentially any user-space vswitch to support *MTS*.

3.2 Detailed Architecture

For the below discussion, we consider the operation of *MTS* for one vswitch compartment and its corresponding tenant VMs from the Level-2 design shown in Fig. 2. The case when only a single compartment (Level-1) is used is similar in vein: the flow table entries installed into the vswitch and the VFs attached to the vswitch compartment need to be modified somewhat; for lack of space we do not detail the Level-1 design any further.

Connectivity. Each vswitch VM is allocated at least one VF (In/Out VF) for external (inter-server) connectivity and another as a gateway (Gw VF) for vswitch-VM-to-tenant-VM connectivity as shown in Fig. 2. Isolation between the external and the tenant network (tenant VF shown as T VF in the Figure) is enforced at the NIC-level by configuring the Gw VF and the tenant VFs with a Vlan tag specific to the tenant. Different Vlan tags are used to further isolate the multiple vswitch compartments and their resp. tenants on that server.

The packets between VFs/PFs in the NIC are forwarded based on the destination MAC address and securely isolated using Vlan tags (the same security model as provided by enterprise Ethernet switches). For all packets to and from the tenant VMs to pass through the vswitch-VM, the destination MAC address of each packet entering and leaving the NIC needs to be accurately set, otherwise packets will not reach the correct destination. This can be addressed by introducing minor configuration changes to the normal operation of the tenant and the vswitches, detailed below.

Ingress chain. Fig. 3ⓐ illustrates the process by which packets from an external network reach the tenant VMs. In step ① a packet enters the server through the NIC fabric

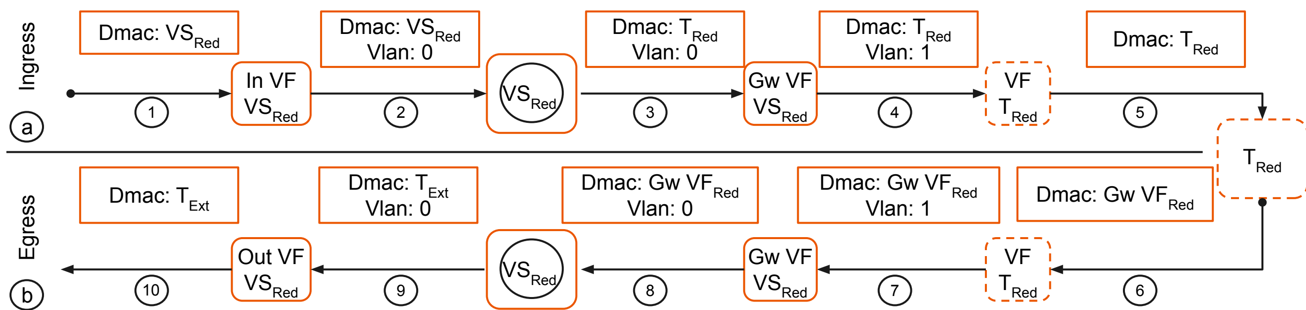


Figure 3: A step-by-step illustration of how packets enter and leave the Red tenant from Figure 2 in *MTS*. ⓐ shows how ingress packets reach Tenant$_{Red}$. ⓑ shows how Tenant$_{Red}$ packets reach an external system Tenant$_{Ext}$.

port having the Red vswitch's In/Out VF MAC address as the destination MAC address (Dmac). The NIC switch will deliver the packet to the vswitch VM untagged (Vlan 0) in ②. The Red vswitch then uses the destination IP address in the packet to identify the correct tenant VM to send the packet to, changes the destination MAC address to that of the Red tenant's VF (VF T_{Red}), and emits the packet to the Gw VF in the NIC in ③. This ensures accurate packet delivery to and from tenant VMs and the complete isolation of the tenant-vswitch traffic from other traffic instances. In ④ and ⑤, the NIC tags the packet with the Red tenant's specific Vlan tag (Vlan 1 in the figure), uses the built-in switch functionality to make a lookup in the MAC learning table for the Vlan, pops the Vlan tag and finally forwards the packet to the Red tenant's VM. The NIC forwarding process is completely transparent to the vswitch and tenant VMs, the only downside is the extra round-trip to the NIC. Later we show that this round-trip introduces negligible latency overhead.

Egress chain. The reverse direction shown in Fig. 3ⓑ, sending a packet from the tenant VM through the vswitch to the external network goes in similar vein. In ⑥ the Red tenant VM T_{Red} sends a packet through its VF (T_{Red}) with the destination MAC address set to the MAC address of the Red tenant's Gw VF; in the next subsection we describe two ways to achieve this. In ⑦ the NIC switch tags the packet (Vlan 1), looks-up the destination MAC address which results in sending the packet to the Gw VF. At the gateway VF ⑧, the NIC switch pops the Vlan tag and delivers the packet to the Red vswitch VM. The vswitch receives the packet, looks up the destination IP address, rewrites the MAC address to the actual (external) gateway's MAC address, and then sends the packet out to the In/Out VF in ⑨. Finally in ⑩, the NIC will in turn send the packet out the physical fabric port.

Communication between the two VMs of a single tenant inside the server goes similarly, with the additional complexity that packets now take *two* extra round-trips to the NIC: once on the way from the sender VM to vswitch, and once on the way from the vswitch to the destination VM. Again, our evaluations in the next sections will show that the induced latency overhead for such a traffic scenario is low.

System support. Next, we detail the modifications the cloud operator needs to apply to the conventional vswitch setup to support *MTS*. The primary requirement is to modify the centralized controllers to appropriately configure tenant specific VFs with Vlan tags and MAC addresses, and insert correct flow rules to ensure the vswitch-tenant connectivity. Second, advanced multi-tenant cloud systems rely on tunneling protocols to support L2 virtual networks. This is also supported by *MTS*, by modifying the flow tables to pop/insert the appropriate headers whenever packets need to be decapsulated/encapsulated. Note that after decapsulation the tunnel id can be used in conjunction with the destination IP address to identify the appropriate tenant VM. Third, the ARP entry for the default gateway must be appropriately set in each tenant VM so that packets from the tenant VM go to the vswitch VM. To this end, the tenant VMs can be configured with a static ARP entry pointing to the appropriate Gw VF, or using the centralized controller and vswitch as a proxy-ARP/ARP-responder [47]. Finally, to prevent malicious tenants from launching an attack on the system, the cloud operator needs to deploy security filters in the NIC. In particular, source MAC address spoofing prevention must be enabled on all tenant VMs' VFs. Furthermore, flow-based wildcard filters can also be applied in the NIC for additional security, e.g., to drop packets not destined to the vswitch compartment, to prevent the Host from receiving packets from the tenant VMs, etc. Our *MTS* implementation, described in Section 4, takes care of removing the manual management burden in applying the above steps.

Resource allocation. Additional levels of security usually come with increased resource requirement, needed to run the security/isolation infrastructure. Below, we describe two resource sharing strategies and how the VFs are allocated to the vswitch compartments. However, due to the sheer quantity and diversity in cloud setups, we restrict the discussion to plain compute and memory resources and the number of SR-IOV VFs for the different *MTS* security levels.

We consider two modes for compute and memory resources. A *shared* mode where tenants' vswitches share a single physical CPU core, while in the *isolated* mode each tenant's vswitch is pinned to a different core. However, we

assume that each vswitch compartment gets an equal share of main memory (ram) and this is inexpensive compared to physical CPU cores. Dedicating compute and memory resources for vswitching is not uncommon among cloud operators [22, 16]. Note that the *shared* and *isolated* resource allocations are merely two ends of the resource allocation spectrum, different sets of vswitch VMs could be allocated resources differently, e.g., based on application or customer requirements. In the next section we will see that the resource requirement for multiple vswitch VM compartments, i.e., Level-2 alone, is not resource prohibitive in the *shared* mode, however, Level-2 and Level-3 can be.

Regarding the number of SR-IOV VFs needed, the current standard allows each SR-IOV device to have up to 64 VFs per PF. For Level-1, the total number of VFs is given by the sum of i) the number of VFs allocated for external connectivity (In/Out VF); ii) the total number of tenant-specific gateway VFs; and iii) tenant-specific VM VFs hosted on the server. In a basic Level-1 setup hosting 1 tenant, with 1 In/Out VF and 1 gateway VF and 1 VF for the tenant VM, the total VFs is 3. Similarly for 4 tenants, the total VFs is 9. For Level-2, the total number of VFs is given by the sum of i) the tenant-specific VFs allocated for external connectivity; ii) the tenant-specific gateway VFs; and iii) tenant-specific VM VFs hosted on the server. For a basic Level-2 setup hosting 2 tenants, with 1 In/Out VF, 1 gateway VF per tenant vswitch and 1 VF for each tenant VM, the total VFs is 6. Similarly for 4 tenants, the total VFs is 12.

4 Evaluating Tradeoffs

We designed a set of experiments to empirically evaluate the security-performance-resource tradeoff of *MTS*. To this end, we measure *MTS*'s performance for different security levels under different resource allocation strategies, in canonical cloud traffic scenarios [19]. The focus is on throughput and latency performance metrics, and physical cores and memory for resources. In particular, the experiments serve to verify our expectation that our design does not introduce a considerable overhead in performance. However, we do expect the amount of resources consumed to increase; our aim is to quantify this increase in different realistic setups.

Prototype framework. We took a programmatic approach to our design and evaluation, hence, we developed a set of primitives that can be composed to configure *MTS* to conduct all the experiments described in this paper. Hence, as a first step we do not consider complex cloud management systems (CMS) such as OpenStack; this way we can conduct self-contained experiments without the possible interference cause by a CMS. Our framework is written in Python and currently supports OvS and ovs-DPDK as the base virtual switch, Mellanox NIC, and the `libvirt` virtualization framework. Our framework and data are available on-line at the following URL: `https://www.github.com/securedataplane`

Methodology. We chose a set of standard cloud traffic scenarios (see Fig. 4) and a fixed number of tenants (4). For each of those scenarios, we allocated the necessary resources (Sec. 3) and then configured the vswitch either in its default configuration (Baseline) or one of the three security levels (Sec. 2.3). The system was then connected in a measurement setup to measure the one-way forwarding performance. Important details on the topology, resources, security levels and the hardware and software used are described next.

Traffic scenarios. The three scenarios evaluated are shown in Fig. 4. *Physical-to-physical (p2p):* Packets are forwarded by the vswitch from the ingress physical port to the egress. This is meant to shed light on basic vswitch forwarding performance. *Physical-to-virtual (p2v):* Packets are forwarded by the vswitch from one physical port to a tenant VM, and then back from the tenant VM to the other physical port. Compared to the p2p, this will show the overhead to forward to and from the tenant VM. *Virtual-to-virtual (v2v):* Similar to the p2v, however, when the packets return from the tenant to the vswitch, the vswitch sends the packet to another tenant which then sends it back to the vswitch and then out the egress port. This scenario emulates service chains in network function virtualization. Since the path length increases from p2p to p2v to v2v, we expect the latency to increase and the throughput to decrease when going from p2p to p2v to v2v.

Resources. We allocated compute resources in the following two ways. *Shared:* All vswitch compartments share 1 physical CPU core and their associated cache levels. *Isolated:* Each vswitch compartment is allocated 1 physical CPU core and their associated cache levels. In case of the Baseline, we allocated cores proportional to the number of vswitch compartments, e.g., 2 cores to compare with 2 vswitch VMs. For main memory, each VM (vswitch and tenant) was allocated 4 GB of which 1 GB is reserved as one 1GB Huge page. Similarly, for the Baseline, a proportional amount of Huge pages was allocated. When using *MTS*, each vswitch VM was allocated 2 In/Out VFs (1 per physical port), and 2 appropriately Vlan tagged Gw VFs per tenant (1 per physical port). When DPDK was used in Level-3: one physical core needs to allocated for each ovs-DPDK compartment (including the Baseline), hence, only the isolated mode was used; all In/Out, gateway and tenant ports connected to OvS were assigned DPDK ports (in the case of the Baseline, the tenant port type was the dpdkvhostuser-client [18]). All the tenant VMs got two physical cores and two VFs, 1 per port (these are VMs the tenant would use to run her application) so that the forwarding app ($l2fwd$) could run without being a bottleneck.

Security levels and tenants. For each resource allocation mode, we configured our setup either in Baseline or one of the three *MTS* security levels (Section 2.3). In the Baseline and Level-1, there were 4 tenant VMs connected to the

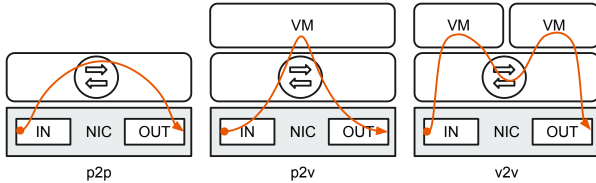


Figure 4: Traffic scenarios evaluated.

vswitch. For Level-2, we configured 2 vswitch VMs and each vswitch had 2 tenant VMs, and then we configured 4 vswitch VMs where each vswitch VM had 1 tenant VM. We repeated Level-3 with Baseline, Level-1 and the two Level-2 configurations.

Setup. To accurately measure the one-way forwarding performance (throughput and latency), we used two servers connected to each other via 10G short range optical links. The device under test (DUT) server was an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz with 64 GB of RAM with the IOMMU enabled but hyper-threading and energy efficiency disabled, and a 2x10G Mellanox ConnectX4-LN NIC with adaptive interrupt moderation and irq balancing disabled. The other server was the packet/load generator (LG), sink and monitor, with an Endace Dag 10X4-P card (which gives us accurate and precise hardware timestamps) [20]. The link between the LG and DUT, and DUT and sink were monitored via a passive optical network tap connected to the Dag card. Each receive stream of the Dag card was allocated 4 GB to receive packets. The Host, vswitch VM and tenant VMs used the Linux kernel 4.4.0-116-generic, Mellanox OFED linux driver 4.3-1.0.1.0, OvS-2.9.0 and DPDK 17.11. Libvirt 1.3.1 was used with QEMU 2.5.0. In the tenant VMs, we adapted the DPDK-17.11 $l2fwd$ app to rewrite the correct destination MAC address when using *MTS*, and used the default $l2fwd$ drain-interval (100 microseconds) and burst size (32) parameters. For the Baseline, we used the default linux bridge in the tenant VMs as using DPDK in the tenant without being backed by QEMU and OvS (e.g., dpdkvhostuserclient) is not a recommended configuration [41]. For network performance measurements, we used Endace dag-5.6.0 software tools (dagflood, dagbits, and dagsnap).

4.1 Throughput

Our first performance tradeoff is evaluating the forwarding throughput. This will shed light on the packets per second (pps) processing performance of *MTS* compared to the Baseline. It also uncovers packet loss sooner than measuring the bandwidth [32]. We measure the aggregate throughput with a constant stream of 64 B packets replayed at line rate (14 Mpps) by the LG and collected at the sink. Since we fixed the number of tenants to 4, the stream of packets comprises 4 flows, each to a respective tenant VM identified by the destination MAC and IP address. At the monitor we collect the packets forwarded to report the aggregate throughput. Each experimental run lasts for 110 seconds and measurements are made from the 10-100 second marks.

Results. The throughput measurement data for the shared mode is shown in Fig. 5(a). In Fig. 5(d) we can see the data for the isolated mode and in Fig. 5(g) the data for Level-3 in the isolated mode is shown. From Figures 5(a) and (d) we can see that nearly always MTS *had either the same or higher aggregate throughput than the Baseline*. The improvement in throughput is most obvious in the p2v and v2v topologies as vswitch-to-tenant communication is via the PCIe bus and NIC switch, which turns out to be faster than Baseline's memory bus and software approach. Sharing the physical core for multiple compartments (Fig. 5(a)) in the p2v and v2v scenarios can offer 4x isolation (Level-2 with 4 compartments) and a 2x increase in throughput (nearly .4 Mpps and .2 Mpps) compared to the Baseline (nearly .2 Mpps and .1 Mpps).

Fig. 5(d) is noteworthy as multiple cores for vswitch VMs and the Baseline functions as a load-balancer when isolating the CPU cores. In the p2p scenario, the aggregate throughput increases roughly from 1 Mpps to 2 Mpps to 4 Mpps as the number of cores increase. We observe that *MTS* is slightly more than the Baseline in the p2p, however, in the p2v and v2v scenarios *MTS* offers higher aggregate throughput. As expected, using DPDK can offer an order of magnitude better throughput (Fig. 5(g)). In the p2p topology, we were able to nearly reach line rate (14.4 Mpps) with four DPDK compartments as the packets were load-balanced across the multiple vswitch VMs, while the Baseline was able to saturate the link with 2 cores. With *MTS*, the throughput saturates (at around 2.3 Mpps) in the p2v and v2v topologies because several ports are polled using a single core and packets have to bounce off the NIC twice as much compared to the Baseline where we observe nearly twice the throughput for 2 and 4 cores. Nevertheless, we can see a slight increase in the throughput of *MTS* as the vswitch VMs increase, because the number of ports per vswitch VM decreases as the number of vswitch VMs increase. Due to the limited physical cores on the DUT, we could not evaluate 4 vswitch VMs in the v2v topology as it required more cores and ram than available.

Key findings. The key result here is that *MTS* offers increasing levels of security with comparable, if not increasing levels of throughput in the shared and isolated resource modes, however, the Baseline's throughput with user-space packet processing (DPDK) is better than *MTS*.

4.2 Latency

The second performance tradeoff we evaluated was the forwarding latency, in particular, we studied the impact of packet size on forwarding. We selected 64B (minimum IPv4 UDP packet size), 512B (average packet), 1500B (maximum MTU) packets and 2048B packets (small jumbo frame). As

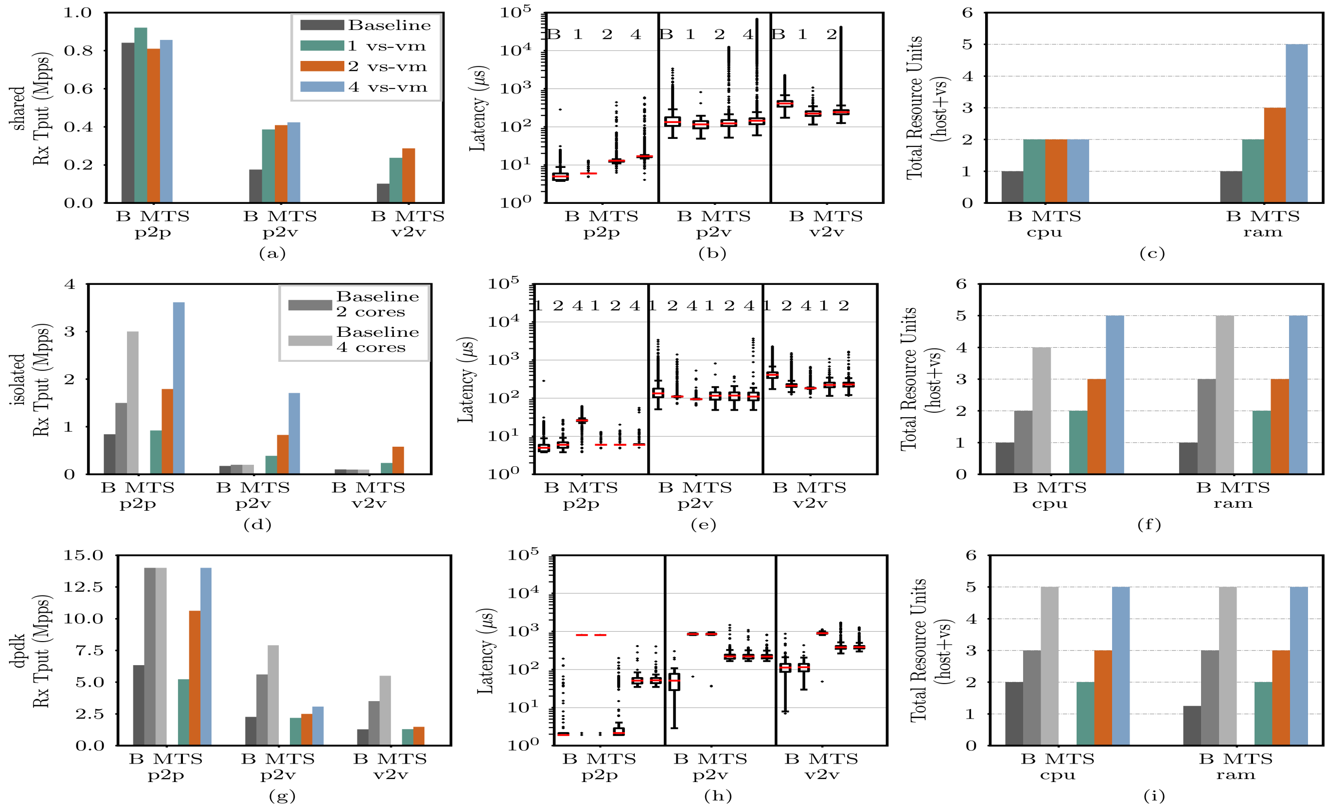


Figure 5: The security, throughput, latency and resource tradeoff comparison of *MTS*. The rows indicate the resource mode. The columns are ordered as throughput, latency and resources. The security levels used are shown in the legend. Note the bottom row is for security Level-3 in the isolated resource mode combined with other security levels.

in the throughput experiments, we used 4 flows, one to each tenant. For each experimental run, we continuously sent packets from the LG to the sink via the DUT at 10 kpps for 30 seconds. Note that is the aggregate throughput sent to the NIC and not to the vswitch VM. To eliminate possible warm-up effects, we only evaluated the packets from the 10-20 second mark.

Results. For brevity the latency distribution only for 64 *B* packets is reported here. Fig. 5(b) shows the data for the shared mode, while Fig. 5(e) is for the isolated mode. Level-3 latency data is shown in Fig. 5(h). Although the p2p scenarios shows that *MTS* increases the latency (Fig. 5(b), (e) and (h)), the p2v and v2v scenarios show that *MTS* is slightly faster than the Baseline. This is for two reasons. First, packets between the vswitch and the tenant VMs pass through the SR-IOV NIC (PCIe bus) rather than a software only vswitch (memory bus). Second, when using the Baseline the tenant uses the Linux bridge. The exception to this can be seen with user-space packet processing (Fig. 5(h)), where the Baseline with a single core for dpdk (2 in total) is always faster than *MTS*. As mentioned in Section. 4.1, due to resource limitations we could not evaluate the 4 vswitch VMs in v2v.

The variance in latency increases as more compartments share the same physical core (Fig. 5(b)). Isolating the vswitch VM cores leads to more predictable latency as seen in Fig. 5(e). When using DPDK (Fig. 5(h)) we make two observations: i) *MTS* takes longer to forward packets than without using DPDK; ii) the latency for Baseline with 2 and 4 cores for dpdk (3 and 5 in total) is unexpectedly high (around 1 ms). Regarding the former, we conclude that *MTS* with OvS and DPDK requires further tuning as we used the default OvS-DPDK parameters for the drain interval, batch size and huge pages: There is an inherent tradeoff between high throughput and average per-packet latency when using a shared memory model where a core is constantly polling [17]. For the latter, we observe that the throughput of 10 kpps is too low to drain the multiple queues on the DPDK ports. At 100 kpps and 1 Mpps, we measured an approximately 2 microsecond latency for the p2p scenario.

Key findings. We observe that for the shared mode, and 4x compartmentalization (Level-2), the latency is comparable to the Baseline (p2v) with a lot of variance whereas when isolated the latency is more predictable.

4.3 Resources

In Fig. 5(c), (f) and (i) we see the total CPU and memory consumption for Baseline and *MTS*. Note that across all the

figures, one core and at least one Huge page is always dedicated for the Host OS. In the case of the (single core) Baseline, the vswitch (OvS) runs in the Host OS and hence shares the Host's core and ram. However, for the single vswitch VM in the shared, isolated and DPDK modes, the Host OS consumes one core and the vswitch VM consumes another core making the total CPU cores two. Similarly, the 2 and 4 vswitch VMs in the shared mode, also consume the same number of cores as the single vswitch VM but a linear increase in ram. In the isolated mode, *MTS* consumes only one extra physical core relative to the Baseline, and in DPDK, *MTS* and Baseline consume equal number of cores. With respect to the memory consumption, we note that *MTS*'s and Baseline's memory consumption in the isolated and DPDK modes are the same.

Hence, we conclude that for one extra physical core, *MTS* offers multiple compartments, making the shared resource allocation economically attractive. The resource cost goes up when user-space packet processing is introduced or isolating cores, making it relatively expensive for multiple vswitch VMs.

Key findings. (i) High levels (2x/4x) of virtual network isolation per server can be achieved with an increase in aggregate throughput (2x) in the shared mode; (ii) for applications that require low and predictable latency, vswitch compartments should use the isolated mode; (iii) although user-space packet processing using DPDK offers high throughput, it is expensive (physical CPU and energy costs).

5 Workload-based Evaluation

We also conducted experiments with real workloads, to gain insights on how cloud applications such as web servers and key-value stores will perform as tenant applications are the end hosts of the virtual networks.

Methodology. For simplicity we focus our workload-based evaluation only on TCP applications as our previous measurements dealt with UDP. In general, we use a similar methodology to the one described in Section 4. For all the TCP-based measurements, we configured the tenant VMs to run the respective TCP server and from the client (LG) we benchmark the server to measure the throughput and/or response time. The topologies, resources and setup used to make these measurements are slightly nuanced which we highlight next.

Traffic scenarios. Only the p2v and v2v patterns are evaluated with workloads as we want to understand the performance of applications hosted in the server.

Resources. The ingress and egress ports for all the traffic are on the same physical NIC port unlike in the previous section where the ingress and egress ports were on separated physical ports of the NIC. Hence, each tenant's vswitch VM was given 1 VF for In/Out and 1 tagged Gw VF. Each tenant VM was given 1 VF.

Setup. The applications generating the load are standard TCP, Apache and Memcached benchmarking tools respectively Iperf3 v3.0.11 [31], ApacheBench v2.3 (ab) [2] and libMemcached v1.0.15 (memslap) [43]. Instead of the Endace card we used a similar Mellanox card at the LG.

5.1 Workloads and Results

Iperf: To compare the maximum achievable TCP throughput, we ran Iperf clients for 100 s with a single stream from the LG to the respective Iperf servers in the DUT's tenant VM. The aggregate throughput was then reported as the sum of throughput for each client-server. We collected 5 such measurements for each experimental configuration and report the mean with 95% confidence.

Webserver: To study workloads from webservers (a very common cloud application), we consider the open-source Apache web server. Using the ApacheBench tool from the LG, we benchmarked the respective tenant webservers by requesting a static 11.3 KB web page from four clients (one for each webserver). Each client made up to 1,000 concurrent connections for 100 s after which we collected the throughput and latency statistics reported by ApacheBench. In the v2v scenario, we used only two client-servers as one of the tenant VMs simply forwarded packets using the DPDK $l2fwd$ app. We collected 5 such repetitions to finally report the average throughput and latency for each experimental configuration with 95% confidence.

Key-value store: Key-value stores are also commonly used cloud applications (e.g., with with webservers). We opted for the open-source Memcached key-value store as it also has an open-source benchmarking tool libMemcached-memslap. We used the default Set/Get ratio of 90/10 for the measurements. The methodology and reporting of the measurements are the same as the webserver.

Results. The data for the Iperf measurements in the shared mode is shown Fig. 6(a). The data for the isolated mode is shown in Fig. 6(f) and Fig. 6(k) depicts the throughput for Level-3. As seen in Section 4.1, here too we observe that *MTS* has a higher throughput (more than 2x in the shared mode) than the Baseline except when DPDK is used in the v2v topology. *MTS* saturated the 10G link in the p2v scenario when isolated and DPDK modes were used.

The data from the throughput measurements for the Apache webserver and Memcached key-value store are first reported in the shared mode in Fig. 6(b) and (c) respectively. For the isolated mode they are shown in Fig. 6(g) and (h). Level-3 throughput is shown in Fig. 6(l) and (m). The three main results from the throughput measurements for Apache and Memcached are the following. *MTS* can offer nearly 2x throughput and 4x isolation (Level-2) in the shared mode. Apache's and Memcached's throughput saturated with *MTS*: we expected the throughput to increase as the vswitch VMs increase when the compartments have isolated cores, how-

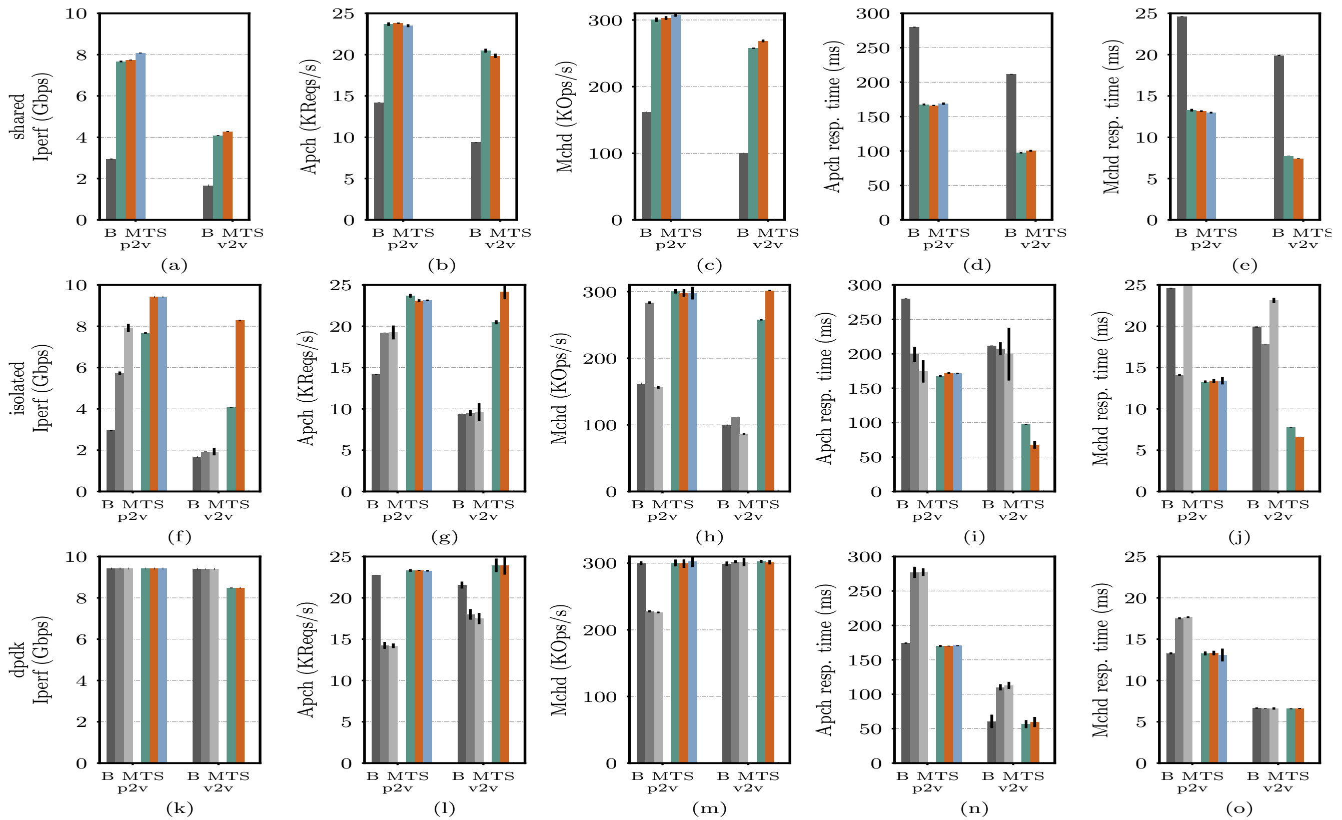


Figure 6: Iperf throughput, Apache and Memcached throughput and latency (shown in the columns) comparison of *MTS*. The rows indicate the resource mode where the bottom row is for security Level-3 in the isolated resource mode combined with the other security levels. The legend is the same as in Figure 5.

ever, we do not observe that. This is further validated when using DPDK. Apache's and Memcached's throughput are highly sensitive when the Baseline uses multiple cores in the isolated and DPDK modes which means that using 2 or more cores requires workload specific tuning to the Host: the DPDK parameters, e.g., drain interval, and the workload VMs, e.g., allocating more cores, which may not always be necessary with *MTS*.

The data from the response time measurements for the Apache webserver and Memcached key-value store are first reported in the shared mode in Fig. 6(d) and (e) respectively. For the isolated mode they are shown in Fig. 6(i) and (j). Level-3 throughput is shown in Fig. 6(n) and (o). Regarding the latency, we again discern that *MTS* can offer multiple levels of isolation and maintain a lower response time (approximately twice as fast) than the Baseline.

Key findings. Our webserver and key-value store benchmarks reveal that application throughput and latency of real application are improved by *MTS*. However, for user-space packet processing, the resource costs go up for a fractional benefit in throughput or latency. Hence, biting the bullet for shared resources, offers 4x isolation and approximately 1.5-2x application performance compared to the Baseline.

6 Discussion

Centralized control, accounting and monitoring. *MTS* introduces the possibility to realize multi-tenant virtual networks which can expose tenant/compartment specific interfaces to a logically centralized control/management plane. This opens up possibilities for full network virtualization, how to expose the interface, and also how to integrate *MTS* into existing cloud management systems in an easy and usable way. Furthermore, controllers may need to manage more device, topology and forwarding information, however, the computations (e.g., routing) should remain the same. From an accounting and billing perspective, we strongly believe that *MTS* is a new way to bill and monitor virtual networks at granularity more than a simple flow rule [24]: CPU, memory and I/O for virtual networking can be charged.

SR-IOV: a double-edged sword. If an attacker can compromise SR-IOV, she could violate isolation and in the worst case get access to the Host OS via the PF driver. Hence, a rigorous security analysis of the SR-IOV standard, implementations and SR-IOV-NIC drivers can reduce the chance of a security vulnerability. Compartmentalizing the PF driver is a promising approach [10]. Furthermore, when a vswitch VM is shared among tenants, performance isolation issues

could lead to covert channels [6] or denial-of-service attacks [64, 75]. Although not yet widely supported, VM migration with SR-IOV can be introduced [45]. SR-IOV NICs have limited VFs and MAC addresses which could limit the scaling properties of *MTS*, e.g., when using containers as compartments instead of VMs.

Evaluation limitations. The results from our experiments are from a network and application performance perspective using a 10 Gbps NIC. For a deeper understanding of the performance improvement we obtained in this paper using SR-IOV, further measurements are necessary, e.g., using the performance monitoring unit (PMU) to collect a breakdown of the packet processing latencies. Such an understanding is important and relevant when dealing with data center applications that require high NIC bandwidth, e.g., 40/100 Gbps.

As described by Neugebauer et al. [48], the PCIe bus can be a bottleneck for special data center applications (e.g., ML applications): A *typical* x8 PCIe 3.0 NIC (with a maximum payload size of 256 bytes and maximum read request of 4096 bytes) has an effective (usable) bi-directional bandwidth of approximately 50 Gbps. Hence, the usability of *MTS* with PCIe 3.0 and 8 lanes can indeed be a limitation which we did not observe in this paper. Nevertheless, increasing the lanes to x16 is one potential workaround to double the effective bandwidth to around 100 Gbps. Furthermore, with chip vendors initiating PCIe 4.0 devices [11], the PCIe bus bandwidth will increase to support intense I/O applications.

7 Related Work

There has been noteworthy research and development on isolating multi-tenant virtual networks in cloud (datacenter) networks: tunneling protocols have been standardized [73, 25], multi-tenant datacenter architectures have been proposed [38], and real cloud systems have been built by many companies [22, 16]. However, most of the previous work still co-locates the vswitch with the Host as we discussed in Section 2.1. Hence, here we discuss previous and existing attempts specifically addressing the security weakness of vswitches.

To the best of our knowledge, in 2012 Jin et al. [34] (see Research prototype in Table 1) were the first to point out the security weakness of co-locating the virtual switch with the hypervisor. However, the proposed design, while ahead of its time, (i) lacks a principled approach which this paper proposes; (ii) has only a single vswitch VM whereas *MTS* supports multiple vswitch compartments making it more robust; (iii) is resource (compute and memory) intensive as the design used shared memory between the vswitch VM and all the tenant VMs while *MTS* uses an inexpensive interrupt-based SR-IOV network card for complete mediation of tenant-vswitch-VM and tenant-host networking; (iv) requires considerable effort, expertise and tuning to integrate into virtualization system whereas *MTS* can easily be scripted into existing cloud systems.

In 2014 Stecklina [65] followed up on this work and proposed sv3, a user-space switch, which can enable multi-tenant virtual switches (see sv3 in Table 1). sv3 adopts user-space packet processing and also supports compartmentalization, i.e., the Host can run multiple vswitches. However, it is still co-located with the Host, partially adopts the security principles outlined in this paper, lacks support for *real* cloud virtual networking, and requires changes to QEMU. Our system on the other hand moves the vswitch out of the Host, supports production vswitches such as OvS and does not require any changes to QEMU.

Between 2016 and 2017, Panda et al. [51] and Neves et al. [49] took a language-centric approach to enforce data plane isolation for virtual networks. However, language-centric approaches require existing vswitches to be reprogrammed/annotated which reduces adoption. Hence the solution of using compartments and SR-IOV in *MTS* allows existing users to easily migrate using their existing software. Shahbaz et al. [63] reduced the attack surface of OvS by introducing support for the P4 domain specific language which reduces potentially vulnerable protocol parsing logic.

In 2018, Pettit et al. [53] proposed to isolate virtual switch packet processing using eBPF: which is conceptually isolating potentially vulnerable parsing code in a kernel-based runtime environment. However, the design still co-locates the virtual switch and the runtime with the Host.

8 Conclusion

This paper was motivated by the observation that while vswitches have been designed to enable multi-tenancy, today's vswitch designs lack strong isolation between tenant virtual networks. Accordingly, we presented a novel vswitch architecture which extends the benefits of multi-tenancy to the virtual switch, offering improved isolation and performance, at a modest additional resource cost. When used in the *shared* mode (only one extra core), with four vswitch compartments the forwarding throughput (in pps) is 1.5-2 times better than the Baseline. The tenant workloads (webserver and key-value stores) we evaluated also receive a 1.5-2 times performance (throughput and response time) improvement with *MTS*.

We believe that *MTS* is a pragmatic solution that can enhance the security and performance of virtual networking in the cloud. In particular, *MTS* introduces a way to schedule an entire core for tenant-specific network virtualization which has three benefits: (i) application and packet processing performance is improved; (ii) this could be integrated into pricing models to appropriately charge customers on-demand and generate revenue from virtual networking for example; (iii) virtual network and Host isolation is maintained even when the vswitch is compromised.

9 Acknowledgments

We thank our shepherd Leonid Ryzhyk and our anonymous reviewers for their valuable feedback and comments. We thank Ben Pfaff, Justin Pettit, Marcel Winandy, Hagen Woesner, Jean-Pierre Seifert and the Security in Telecommunications (SecT) team from TU Berlin for valuable discussions at various stages of this paper. The first author (K. T.) acknowledges the financial support by the Federal Ministry of Education and Research of Germany in the framework of the Software Campus 2.0 project nos. 01IS17052 and 01IS1705, and the "API Assistant" activity of EIT Digital. The third author (G. R.) is with the MTA-BME Information Systems Research Group.

References

[1] ALLCLAIR, T. Secure Container Isolation: Problem Statement & Solution Space. https://docs.google.com/document/d/1QQ5u1RBDLXWvC8K3pscTtTRThsOeBSts_imYEoRyw8A, 2018. Accessed: 05-01-2019.

[2] APACHE. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.2/en/programs/ab.html. Accessed: 07-01-2019.

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure linux containers with intel SGX. In *Proc. Usenix Operating Systems Design Principles (OSDI)* (2016).

[4] AWS. Enhanced Networking on Linux. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html, 2018. Accessed: 24-01-2018.

[5] AZURE, M. Create a Linux virtual machine with Accelerated Networking. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli, 2018. Accessed: 24-01-2018.

[6] BATES, A., MOOD, B., PLETCHER, J., PRUSE, H., VALAFAR, M., AND BUTLER, K. On detecting co-resident cloud instances using network flow watermarking techniques. *Springer International Journal of Information Security* (2014).

[7] BEAUPRÉ, A. Updates in container isolation. https://lwn.net/Articles/754433, 2018. Accessed: 09-01-2019.

[8] BESS COMITTERS. BESS (Berkeley Extensible Software Switch). https://github.com/NetSys/bess, 2017. Accessed: 09-05-2017.

[9] BISHOP, M. A. *Introduction to computer security*, vol. 50. Addison-Wesley Boston, 2005.

[10] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating malicious device drivers in linux. In *Proc. Usenix Annual Technical Conference (ATC)* (2010).

[11] Broadcom Samples Thor, World's First 200G Ethernet Controller with 50G PAM-4 and PCIe 4.0. https://www.broadcom.com/company/news/product-releases/2367107. Accessed: 06-05-2019.

[12] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do: Security and Functionality in a Commodity Hypervisor. In *Proc. ACM Symposium on Operating System Principles (SOSP)* (2011).

[13] COLUMBUS, L. Roundup Of Cloud Computing Forecasts And Market Estimates. https://www.forbes.com/sites/louiscolumbus/2018/09/23/roundup-of-cloud-computing-forecasts-and-market-estimates-2018/, 2017. Accessed: 09-01-2019.

[14] COSTIN, A. All your cluster-grids are belong to us: Monitoring the (in)security of infrastructure monitoring systems. In *Proc. IEEE Communications and Network Security (CNS)* (Sept 2015).

[15] CSIKOR, L., ROTHENBERG, C., PEZAROS, D. P., SCHMID, S., TOKA, L., AND RÉTVÁRI, G. Policy injection: A cloud dataplane dos attack. In *Proc. ACM SIGCOMM Posters and Demos* (2018).

[16] DALTON, M., SCHULTZ, D., ADRIAENS, J., AREFIN, A., GUPTA, A., FAHS, B., RUBINSTEIN, D., ZERMENO, E. C., RUBOW, E., DOCAUER, J. A., ALPERT, J., AI, J., OLSON, J., DECABOOTER, K., DE KRUIJF, M., HUA, N., LEWIS, N., KASINADHUNI, N., CREPALDI, R., KRISHNAN, S., VENKATA, S., RICHTER, Y., NAIK, U., AND VAHDAT, A. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2018).

[17] DPDK. Writing Efficient Code. https://doc.dpdk.org/guides/prog_guide/writing_efficient_code.html. Accessed: 06-01-2019.

[18] [PATCH] netdev-dpdk: Add new 'dpdkvhostuser-client' port type. https://mail.openvswitch.org/pipermail/ovs-dev/2016-August/321742.html. Accessed: 29-04-2019.

[19] EMMERICH, P., RAUMER, D., WOHLFART, F., AND CARLE, G. Performance characteristics of virtual switching. In *Proc. IEEE Conference on Cloud Networking* (2014).

[20] Endace DAG 10X4-P Datasheet. https://www.endace.com/dag-10x4-p-datasheet.pdf. Accessed: 07-01-2019.

[21] FIRESTONE, D. Vfp: A virtual switch platform for host sdn in the public cloud. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2017), pp. 315–328.

[22] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., 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. Azure accelerated networking: Smartnics in the public cloud. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2018).

[23] FRAZELLE, J. Hard multi-tenancy in kubernetes. https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes, 2018. Accessed: 09-01-2019.

[24] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing#network, 2018. Accessed: 03-01-2019.

[25] Geneve: Generic Network Virtualization Encapsulation. https://tools.ietf.org/html/draft-ietf-nvo3-geneve-08. Accessed: 03-01-2019.

[26] The gVisor project. https://github.com/google/gvisor, 2018. Accessed: 09-01-2019.

[27] GOSPODAREK, A. The Rise of SmartNICs – offloading dataplane traffic to...software. https://youtu.be/AGSy51VlKaM, 2017. Open vSwitch Conference.

[28] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mswitch: a highly-scalable, modular software switch. In *Proc. ACM Symposium on SDN Research (SOSR)* (2015).

[29] HWANG, J., RAMAKRISHNAN, K., AND WOOD, T. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2014).

[30] INTEL. Enabling NFV to deliver on its promise. https://www-ssl.intel.com/content/www/us/en/communications/nfv-packet-processing-brief.html, 2015.

[31] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/. Accessed: 07-01-2019.

[32] JACOBSON, V. Congestion avoidance and control. In *ACM Computer Communication Review (CCR)* (1988).

[33] JAIN, R., AND PAUL, S. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communication Magazine 51*, 11 (2013).

[34] JIN, X., KELLER, E., AND REXFORD, J. Virtual switching without a hypervisor for a more secure cloud. In *Proc. USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (HotICE)* (2012).

[35] JING, C. Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution. http://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution_593986, 2018. Accessed: 03-01-2019.

[36] The Kata Containers project. https://katacontainers.io, 2018. Accessed: 09-01-2019.

[37] KO, M., AND RECIO, R. Virtual ethernet bridging. http://www.ieee802.org/1/files/public/docs2009/new-hudson-vepa_seminar-20090514d.pdf. Accessed: 06-01-2019.

[38] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2014).

[39] KUTCH, P. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note* (2011), 321211–002.

[40] LÉVAI, T., PONGRÁCZ, G., MEGYESI, P., VÖRÖS, P., LAKI, S., NÉMETH, F., AND RÉTVÁRI, G. The Price for Programmability in the Software Data Plane: The Vendor Perspective. *IEEE J. Selected Areas in Communications* (2018).

[41] HowTo Launch VM over OVS-DPDK-17.11 Using Mellanox ConnectX-4 and ConnectX-5. `https://community.mellanox.com/s/article/howto-launch-vm-over-ovs-dpdk-17-11-using-mellanox-connectx-4-and-connectx-5`. Accessed: 09-01-2019.

[42] MELLANOX. Mellanox BlueField SmartNIC. `https://bit.ly/2JaMitA`, 2017. Accessed: 05-06-2018.

[43] Memcached. `https://libmemcached.org/libMemcached.html`. Accessed: 07-01-2019.

[44] MICROSOFT. Hyper-V Virtual Switch Overview. `https://technet.microsoft.com/en-us/library/hh831823(v=ws.11).aspx`, 2013. Accessed: 27-01-2017.

[45] MICROSOFT. SR-IOV VF Failover and Live Migration Support. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/sr-iov-vf-failover-and-live-migration-support`, 2017. Accessed: 03-01-2019.

[46] MOLNÁR, L., PONGRÁCZ, G., ENYEDI, G., KIS, Z. L., CSIKOR, L., JUHÁSZ, F., KŐRÖSI, A., AND RÉTVÁRI, G. Dataplane specialization for high-performance openflow software switching. In *Proc. ACM SIGCOMM* (2016).

[47] MULLER, A. OVS ARP Responder – Theory and Practice. `https://assafmuller.com/2014/05/21/ovs-arp-responder-theory-and-practice/`. Accessed: 06-01-2019.

[48] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding pcie performance for end host networking. In *Proc. ACM SIGCOMM* (2018).

[49] NEVES, M., LEVCHENKO, K., AND BARCELLOS, M. Sandboxing data plane programs for fun and profit. In *Proc. ACM SIGCOMM Posters and Demos* (2017).

[50] PALADI, N., AND GEHRMANN, C. Sdn access control for the masses. *Elsevier Computers & Security* (2019).

[51] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. Netbricks: Taking the v out of nfv. In *Proc. Usenix Operating Systems Design Principles (OSDI)* (2016).

[52] PANICKER, M. Enabling Hardware Offload of OVS Control & Data plane using LiquidIO. `https://youtu.be/qjXBRCFhbqU`, 2017. Open vSwitch Conference.

[53] PETTIT, J., PFAFF, B., STRINGER, J., TU, C.-C., BLANCO, B., AND TESSMER, A. Bringing platform harmony to vmware nsx. *ACM SIGOPS Operating System Review* (2018).

[54] PETTIT, J., PFAFF, B., WRIGHT, C., AND VENUGOPAL, M. OVN, Bringing Native Virtual Networking to OVS. `https://networkheresy.com/2015/01/13/ovn-bringing-native-virtual-networking-to-ovs/`, 2015. Accessed: 27-01-2017.

[55] PFAFF, B. Open vSwitch: Past, Present, and Future. `http://openvswitch.org/slides/ppf.pdf`, 2013. Accessed: 27-01-2017.

[56] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of Open vSwitch. In *Proc. Usenix Networked Systems Design and Implementation (NSDI)* (2015).

[57] RAM, K. K., COX, A. L., CHADHA, M., RIXNER, S., AND BARR, T. Hyper-switch: A scalable software virtual switching architecture. In *Proc. Usenix Annual Technical Conference (ATC)* (2013).

[58] RIZZO, L. Netmap: a novel framework for fast packet I/O. In *Proc. Usenix Annual Technical Conference (ATC)* (2012).

[59] RIZZO, L., AND LETTIERI, G. VALE, a switched ethernet for virtual machines. In *Proc. ACM CoNEXT* (2012).

[60] ROBIN G. Open vSwitch with DPDK Overview. `https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview`, 2016. Accessed: 27-01-2017.

[61] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[62] SECURITYTWEEK. CSA's cloud adoption, practices and priorities survey report. `http://www.securityweek.com/data-security-concerns-still-challenge`, 2015. Accessed: 09-01-2019.

[63] SHAHBAZ, M., CHOI, S., PFAFF, B., KIM, C., FEAMSTER, N., MCKEOWN, N., AND REXFORD, J. Pisces: A programmable, protocol-independent software switch. In *Proc. ACM SIGCOMM* (2016).

[64] SMOLYAR, I., BEN-YEHUDA, M., AND TSAFRIR, D. Securing self-virtualizing ethernet devices. In *Proc. Usenix Security Symp.* (2015).

[65] STECKLINA, J. Shrinking the hypervisor one subsystem at a time: A userspace packet switch for virtual machines. In *Proc. ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)* (2014).

[66] STONE, R. PCI SR-IOV on FreeBSD. https://people.freebsd.org/~rstone/BSDCan_SRIOV.pdf. Accessed: 06-01-2019.

[67] THE FAST DATA PROJECT. What is the Fast Data Project (FD.io)? https://fd.io/about, 2017. Accessed: 05-06-2018.

[68] THIMMARAJU, K., RÉTVÁRI, G., AND SCHMID, S. Virtual network isolation: Are we there yet? In *Proc. ACM Workshop on Security in Softwarized Networks: Prospects and Challenges* (2018).

[69] THIMMARAJU, K., SHASTRY, B., FIEBIG, T., HETZELT, F., SEIFERT, J.-P., FELDMANN, A., AND SCHMID, S. Taking control of sdn-based cloud systems via the data plane. In *Proc. ACM Symposium on SDN Research (SOSR)* (2018).

[70] TSENG, J., ET AL. Accelerating open vswitch with integrated gpu. In *Proc. ACM Workshop on Kernel-Bypass Networks* (2017).

[71] VANOVER, R. Virtual switching to become enhanced with Cisco and VMware announcement. http://www.techrepublic.com/blog/data-center/virtual-switching-to-become-enhanced-with-cisco-and-vmware-announcement, 2008. Accessed: 27-01-2017.

[72] VMWARE. VMware ESX 4.0 Update 1 Release Notes. https://bit.ly/2sFTuTy, 2009. Accessed: 05-06-2018.

[73] Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 network. https://tools.ietf.org/html/rfc7348. Accessed: 01-06-2016.

[74] ZHAO, Y. PCI: Linux kernel SR-IOV support. https://lwn.net/Articles/319651/, 2009. Accessed: 06-01-2019.

[75] ZHOU, Z., LI, Z., AND ZHANG, K. All your vms are disconnected: Attacking hardware virtualized network. In *Proc. ACM Conference on Data and Application Security and Privacy (CODASPY)* (2017).

StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone

Heejin Park[1], Shuang Zhai[1], Long Lu[2], and Felix Xiaozhu Lin[1]

[1]Purdue ECE [2]Northeastern University

Abstract

While it is compelling to process large streams of IoT data on the cloud edge, doing so exposes the data to a sophisticated, vulnerable software stack on the edge and hence security threats. To this end, we advocate isolating the data and its computations in a trusted execution environment (TEE) on the edge, shielding them from the remaining edge software stack which we deem untrusted.

This approach faces two major challenges: (1) executing high-throughput, low-delay stream analytics in a single TEE, which is constrained by a low trusted computing base (TCB) and limited physical memory; (2) verifying execution of stream analytics as the execution involves untrusted software components on the edge. In response, we present StreamBox-TZ (SBT), a stream analytics engine for an edge platform that offers strong data security, verifiable results, and good performance. SBT contributes a data plane designed and optimized for a TEE based on ARM TrustZone. It supports continuous remote attestation for analytics correctness and result freshness while incurring low overhead. SBT only adds 42.5 KB executable to the TCB (16% of the entire TCB). On an octa core ARMv8 platform, it delivers the state-of-the-art performance by processing input events up to 140 MB/sec (12M events/sec) with sub-second delay. The overhead incurred by SBT's security mechanism is less than 25%.

1 Introduction

Many key applications of Internet of Things (IoT) process a large influx of sensor[1] data, i.e. telemetry. Smart grid aggregates power telemetry to detect supply/demand imbalance and power disturbances [76], where a power sensor is reported to produce up to 140 million samples per day [16,17]; oil producers monitor pump pressure, tank status, and fluid temperatures to determine if wells work at ideal operating points [55,60], where an oil rig is reported to produce 1–2 TB of data per day [43]; manufacturers continuously monitor vibration and ultrasonic energy of industrial equipment for detecting equipment anomaly and predictive maintenance [104, 120], where a monitored machine is reported to generate PBs of data in a few days [77].

[1]Recognizing that IoT data sources range from small sensors to large equipment, we refer to them all as *sensors* for brevity.

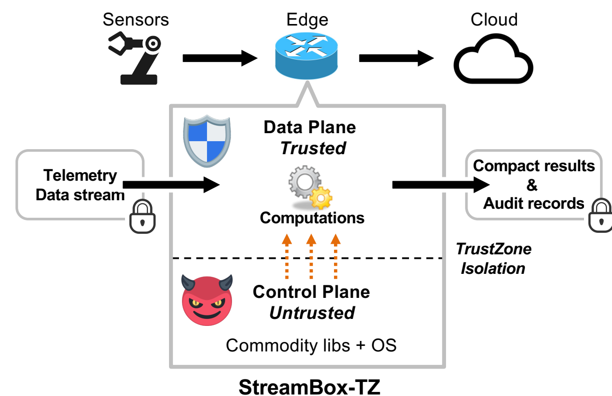


Figure 1: An overview of StreamBox-TZ

The large telemetry data streams must be processed in time. The high cost and long delay in transmitting data necessitate edge processing [98, 100]: sensors send the data to nearby gateways dubbed "cloud edge"; the edge runs a pipeline of continuous computations to cleanse and summarize the telemetry data and reports the results to cloud servers for deeper analysis. Edge hardware is often optimized for cost and efficiency. According to a 2018 survey [45], modern ARM machines are typical choices for edge platforms. Such a platform often has 2–8 CPU cores and several GB DRAM.

Unfortunately, edge processing exposes IoT data to high security threats. i) Deployed in the wild, the edge suffers from common IoT weaknesses, including lack of professional supervision [58, 118], weak configurations [108, 117], and long delays in receiving security updates [58, 114]. ii) On the edge, the IoT data flows through a set of sophisticated components that expose a wide attack surface. These components include a commodity OS (e.g. Linux or Windows), a variety of user libraries, and a runtime framework called *stream analytics engine* [37, 42, 83]. They reuse much code developed for servers and workstations. Their exploitable mis-

configurations [121] and vulnerabilities [23, 35, 109] are not uncommon. iii) With data aggregated from multiple sources, the edge is a high-value target to adversaries. For these reasons, edge is even more vulnerable than sensors, which run much simpler software with narrower attack surfaces. Once attackers compromise the edge, they not only access confidential data but also may delete or fabricate data sent to the cloud, threatening the integrity of an entire IoT deployment.

Towards secure stream analytics on an edge platform, our goal is to safeguard IoT data confidentiality and integrity, support verifiable results, and ensure high throughput with low output delay. Following the principle of least privilege [95], we protect the analytics data and computations in a trusted execution environment (TEE) and limit their interface; we leave out the remaining edge software stack which we deem untrusted. By doing so, we shrink the trusted computing base (TCB) to only the protected functionalities, the TEE, and the hardware. We hence significantly enhance data security.

We face three challenges: i) what functionalities should be protected in TEE and behind what interfaces? ii) how to execute stream analytics on a TEE's low TCB and limited physical memory while still delivering high throughput and low delay? iii) as both trusted and untrusted edge components participate in stream analytics, how to verify the outcome?

Existing solutions are inadequate: pulling entire stream analytics engines to TEE [22, 27, 112] would result in a large TCB with a wide attack surface; the systems securing distributed operators [53, 99, 124] often lack stream semantics or optimizations for efficient execution in a single TEE, which are crucial to the edge; only attesting TEE integrity [65] or data lineages [50, 99, 102, 124] is inadequate for verifying stream analytics. We will show more evidences in the paper.

Our response is StreamBox-TZ (SBT), a secure engine for analyzing telemetry data streams. As shown in Figure 1, SBT builds on ARM TrustZone [2] on an edge platform. SBT contributes the following notable designs:

(1) Architecting a data plane for protection SBT provides a data plane exposing narrow, shared-nothing interfaces to untrusted software. SBT's data plane encloses i) all the analytics data; ii) a new library of low-level stream algorithms called *trusted primitives* as the only allowed computations on the data; iii) key runtime functions, including memory management and cache-coherent parallel execution of trusted primitives. SBT leaves thread scheduling and synchronization out of TEE.

(2) Optimizing data plane performance within a TEE In contrast to many TEE-oblivious stream engines that operate numerous small objects, hash tables, and generic memory allocators [32, 82, 122], SBT embraces unconventional design decisions for its data plane. i) SBT implements trusted primitives with array-based algorithms and contributes new optimizations with handwritten ARMv8 vector instructions. ii) To process high-velocity data in TEE, SBT provides a new abstraction called uArrays, which are contiguous, virtually unbounded buffers for encapsulating all the analytics data; SBT backs uArrays with on-demand paging in TEE and manages uArrays with a specialized allocator. The allocator leverages hints from untrusted software for compacting memory layout. iii) SBT exploits TrustZone's lesser-explored hardware features: ingesting data straightly through trusted IO without a detour through the untrusted OS; avoiding relocating streaming data by leveraging the large virtual address space dedicated to a TEE.

(3) Verifying edge analytics execution SBT supports cloud verifiers to attest analytics *correctness*, result *freshness*, and the untrusted hints received during execution. SBT captures coarse-grained dataflows and generates audit records. A cloud verifier replays the audit records for attestation. To minimize overhead in the edge-cloud uplink bandwidth, SBT compresses the records with domain-specific encoding.

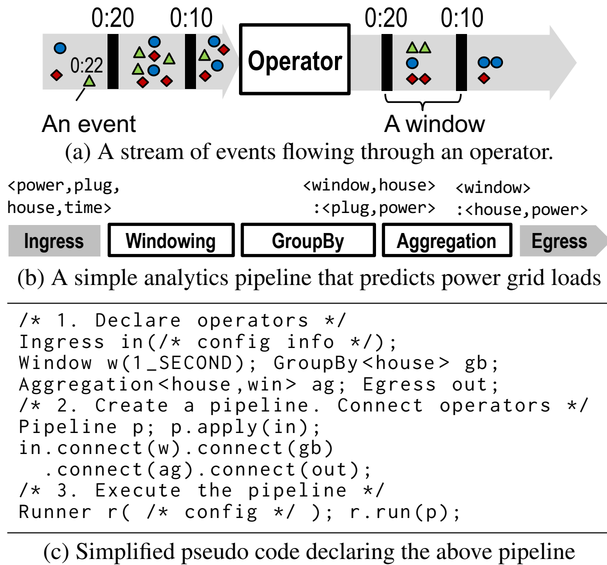
Our implementation of SBT supports a generic stream model [1] with a broad arsenal of stream operators. The TCB of SBT contains as little as 267.5 KB of executable code, of which SBT only constitutes 16%. On an octa core ARMv8 platform, SBT processes up to 12M events (144 MB) per second at sub-second output delays. Its throughput on this platform is an order of magnitude higher than an SGX-based secure stream engine running on a small x86 cluster with richer hardware resources [53]. The security mechanisms contributed by SBT incur less than 25% throughput loss with the same output delay; decrypting ingress data, when needed, incurs 4%–35% throughput loss with the same output delay. While sustaining high throughput, SBT uses up to 130 MB of physical memory in most benchmarks.

The key contributions of SBT are: i) a stream engine architecture with strongly isolated data and a lean TCB; ii) a data plane built from the ground up with computations and memory management optimized for a single TrustZone-based TEE; iii) remote attestation for stream analytics on the edge with domain-specific compression of audit records. To our knowledge, SBT is the first system designed and optimized for data-intensive, parallel computations inside ARM TrustZone. Beyond stream analytics, the SBT architecture should help secure other important analytics on the edge, e.g. machine learning inference. The SBT source can be found at `http://xsel.rocks/p/streambox`.

2 Background & Motivation

2.1 ARM for Cloud Edge

As typical hardware for IoT gateways [45], recent ARM platforms offer competitive performance at low power, suiting edge well. Most modern ARM cores are equipped with TrustZone [2], a security extension for TEE enforcement. TrustZone logically partitions a platform's hardware resources, e.g. DRAM and IO, into a normal (insecure) and a secure world. CPU cores independently switch between two worlds. A TEE



(a) A stream of events flowing through an operator.

(b) A simple analytics pipeline that predicts power grid loads

```
/* 1. Declare operators */
Ingress in(/* config info */);
Window w(1_SECOND); GroupBy<house> gb;
Aggregation<house,win> ag; Egress out;
/* 2. Create a pipeline. Connect operators */
Pipeline p; p.apply(in);
in.connect(w).connect(gb)
  .connect(ag).connect(out);
/* 3. Execute the pipeline */
Runner r( /* config */ ); r.run(p);
```

(c) Simplified pseudo code declaring the above pipeline

Figure 2: Example stream data, operators, and a pipeline

atop TrustZone owns dedicated, *trusted IO*, a unique feature that other TEE technologies such as Intel SGX [81] lack.
Trusted IO is a unique feature of ARM TrustZone, implemented through hardware components including TrustZone Address Space Controller (TZASC) and TrustZone Protection Controller (TZPC). TZASC allows privilege software to logically partition DRAM between the normal and the secure worlds. Similarly, TZPC allows to configure IO peripherals accessible to either world. Any peripheral owned by the secure world is completely enclosed in the secure world. We use trusted IO to support the trusted source-edge links on the cloud edge (§3.1).

2.2 Stream Analytics

Stream Model We target stream analytics over sensor data. A data stream consists of sensor events that carry timestamps defined by event occurrence, as illustrated in Figure 2(a). Programmers specify a pipeline of continuous computations called *operators*, e.g. `Select` and `GroupBy`, that are extensively used for telemetry analytics [62, 90]. As data arrives at the edge, a stream analytics engine ingests the data at the pipeline ingress, pushes the data through the pipeline, and externalizes the results at the pipeline egress.

We follow a generic stream model [14, 32, 69, 85, 122]. Operators execute on event-time scopes called *windows*. Data sources emit special events called *watermarks*. A watermark guarantees no subsequent events in the stream will have event times earlier than the watermark timestamp. A pipeline's *output delay* is defined as the elapsed time starting from the moment the ingress receives the watermark signaling the completion of the current window to the moment the egress externalizes the window results [82]. A pipeline may maintain its internal states organized by windows at different operators. See prior work [20] for a formal stream model.

Analytics example: Power load prediction Figure 2(b-c) shows an example derived from an IoT scenario [62]: it predicts future household power loads based on power loads reported by smart power plugs. The example pipeline ingests a stream of power samples and groups them by 1-second fixed windows and by houses. For each house in each window, it aggregates all the loads and predicts the next-window load as an exponentially weighted moving average over the recent windows. At the egress, the pipeline emits a stream of per-house load prediction for each window.

Stream analytics engines Stream pipelines are executed by a runtime framework called a stream analytics engine [37, 42, 46, 82, 83, 90]. A stream analytics engine consists of two types of function: *data functions* for data move and computations; *control functions* for resource management and computation orchestration, e.g. creating and scheduling tasks. The boundary between the two is often blurry. To amortize overheads, control functions often organize data in *batches* and invoke data functions to operate on the batches.

2.3 Security Threats & Design Objectives

The edge faces common threats in IoT deployment. i) IT expertise is weak. Edge platforms are likely managed by field experts [58, 114, 118] rather than IT experts. Such lack of professional supervision is known to result in weak configurations [108, 117]. ii) The infrastructure is weak. Deployed in the field, the edge often sees slow uplinks [84, 114] and hence much delayed software security updates. For cost saving, edge analytics may need to share OS and hardware with other high-risk, untrusted software such as web browsers [114].

Besides the common threats, existing edge software stacks entrust IoT data with commodity OSes, analytics engines, and language runtimes (e.g. JVM). However, these components are incapable of offering strong security guarantees due to their complexity and wide interfaces. Each of them easily contains more than several hundreds of KSLoC [116]. Exploitable vulnerabilities are constantly discovered [3, 6, 23, 35, 38], making these components untrusted in recent research [36, 54, 79, 80]. By exploiting these vulnerabilities, a local adversary as an edge user program may compromise the kernel through the wide user/kernel interfaces [11, 12] or attack an analytics engine through IPC [7]; a remote adversary, through the edge's network services, may compromise analytics engines [4] or the OS [10]. A successful adversary may expose IoT data, corrupt the data, or covertly manipulate the data. Taking the application in Figure 2(b) as an example, the adversary gains access to the smart plug readings, which may contain residents' private information, and injects fabricated data.

Objectives We aim three objectives for stream analytics over telemetry data on an edge platform: i) confidentiality and integrity of IoT data, raw or derived; ii) verifiable correctness

and freshness of the analytics results; iii) modest security overhead and good performance.

3 Security Approach Overview

3.1 Scope

IoT scenarios We target an edge platform that captures and analyzes telemetry data. We recognize the significance of mission-critical IoT with tight control loops, but do not target it. Our target scenario includes source sensors, edge platforms, and a cloud server which we dub "cloud consumer". All the raw IoT data and analytics results are owned by one party. The sensors produce trusted events, e.g. by using secure sensing techniques [49, 73, 97]. The cloud consumer is trusted; it installs analytics pipelines to the edge and consumes the results uploaded from the edge. We consider *untrusted* source-edge links (e.g. public networks) which requires data encryption by the source, as well as *trusted* source-edge links (e.g. direct IO bus or on-premise local networks), and will evaluate the corresponding designs (§9). We assume untrusted edge-cloud links, which require encryption of the uploaded data.

In-scope Threats We consider malicious adversaries interested in learning IoT data, tampering with edge processing outcome, or obstructing processing progress. We assume powerful adversaries: by exploiting weak configurations or bugs in the edge software, they already control the entire OS and all applications on the edge.

Out-of-scope Threats We do not protect the confidentiality of stream pipelines, in the interest of including only low-level compute primitives in a lean TCB. We do not defend the following attacks. i) Attacks to non-edge components assumed trusted above, e.g. sensors [111]. ii) Exploitation of TEE kernel bugs [8, 9, 56]. iii) Side channel attacks: by observing hardware usage outside TEE, adversaries may learn the properties of protected data, e.g. key skew [72]. Note that controlled-channel attack [119] cannot be applied to ARM TrustZone as it has separate page management within a separate secure OS unlike Intel SGX. iv) Physical attacks, e.g. sniffing TEE's DRAM access [18, 28]. Many of these attacks are mitigated by prior work [39, 66, 123, 124] orthogonal to SBT.

Note that TEE code authenticity and integrity are already ensured by the TrustZone hardware, i.e. only code trusted by the device vendor can run in TrustZone and its integrity is protected by TrustZone.

3.2 Approach and Security Benefits

As shown in Figure 3, SBT protects its data functions in a trusted *data plane* in TEE. SBT runs its untrusted *control plane* in the normal world. The control plane invokes the data plane through narrow, shared-nothing interfaces. The engine's

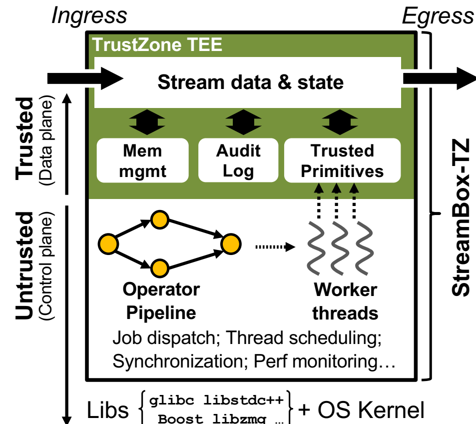


Figure 3: StreamBox-TZ on an edge platform with ARM TrustZone. Bold arrows show the protected data path.

TCB thus only consists of the TEE (including the data plane) and the hardware.

Streaming data always flows in TEE. The data plane ingests the data through TrustZone's trusted IO. After ingestion, it returns *opaque references* of the data batches to the control plane. In turn, the control plane requests computations on the protected data by invoking the data plane with the opaque references. The data plane generates opaque references as long, random integers. It tracks all live opaque references, validates incoming opaque references, and only accepts ones that exist. At the pipeline egress, the data plane encrypts, signs, and sends the result to the cloud.

The analytics execution is continuously attested. SBT captures complete and deterministic dataflows of the stream analytics as well as execution timing, and periodically reports to the cloud server. The cloud server verifies if all ingested data is processed according to the pipeline (correctness), and if the edge incurs low delay (freshness).

Thwarted attacks SBT defeats the following attacks. ***i) Breaking IoT data confidentiality or integrity.*** As the raw and derived data enters and leaves the edge TEE through trusted IOs, adversaries on the edge cannot touch, drop, or inject data. When the data is off the edge transmitted over untrusted networks, it is protected by encryption against network-level adversaries. ii) ***Breaking the data plane integrity.*** Any fabricated opaque reference passed to the data plane will be rejected, since all opaque references are validated before use. Through the data plane's interface, an adversary may exploit bugs in the data plane and compromise it. By minimizing the date plane codebase and hardening its interface, SBT substantially reduces the data plane's attack surface and potential bugs that can be exploited. iii) ***Breaking analytics correctness.*** A compromised control plane may request computations deviating from pipeline declarations or the stream model. For instance, it may invoke trusted computations on partial data, wrong windows, or valid but undesirable opaque references. SBT defeats these attacks through attestation: since the cloud verifier possesses complete knowledge on ingested data and

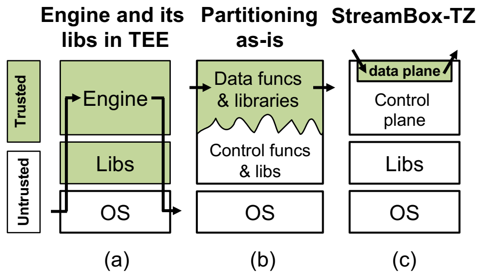


Figure 4: Among alternative architectures for secure stream analytics, StreamBox-TZ (c) leads to the smallest TCB and the most optimized data plane. Arrows indicate data flows.

System	TEE	Analytics	SG	Compute in TEE	Memory	Attestation
VC3 [99]	SGX	Batch	`CIVA-`	Mapper/reducer	Heap	Data lineage
Opaque [124]	SGX	Batch	`CIVAO`	Query plans	unreported	Data lineage
EnclaveDB [91]	SGX	Batch	`CI-A-`	Pre-compiled queries	unreported	TEE integrity
SafeBricks [89]	SGX	Pkt proc.	`CI-A-`	Net func. operators*	unreported	TEE integrity
SecureStream [53]	SGX	Stream	`CI---`	Lua programs	unreported	TEE integrity
StreamBox-TZ	TZ	Stream	`CIV--`	Vectorized primitives*	uArray	Log replay

SG: security guarantees.
`C`: data confidentiality; `I`: data integrity; `V`: verifiability; `A`: analytics confidentiality; `O`: obliviousness
* TEE encloses only low-level computations; otherwise TEE encloses whole analytics.

Table 1: Comparison to existing secure processing systems

Trusted Primitives	Popular Spark Streaming Operators
Sort, Merge, Segment, SumCnt, TopK, Concat, Join, Count, Sum, Unique, FileterBand, Median, ...	GroupByKey, Windowing, AvgPerKey, Distinct, SumByKey, AggregateByKey, SortByKey, TopKPerKey, CountByKey, CountByWindow, Filter, MedianByKey, TempJoin, Union, ...

Table 2: Selected trusted primitives (23 in total) and operators they constitute. These operators cover most listed in the Spark Streaming documentation [103].

pipelines, it detects such correctness violation and rejects the edge analytics results. ***iv) Attacks on analytics performance or availability.*** A compromised control plane may delay or pause invoking of trusted computations, violating the freshness guarantee. As the execution timing of trusted computations is attested, the cloud verifier detects the attacks and can choose to prompt further investigation. ***v) Attempting to trigger data race or deadlock.*** By design, data race and deadlocks will never happen inside the data plane: the trusted computations do not share state concurrently and all locking happens outside of the TEE.

4 Design Overview

4.1 Challenges

Our approach raises three challenges. i) ***Architecting the engine with a proper protection boundary***. This hinges on a key trade-off among TEE functional richness, overhead of TEE entry/exit, and TCB size. ii) ***Optimizing data functions within a TEE***. Processing of high-velocity data in a TEE strongly favors simple algorithms and compact memory. Yet, existing stream engines often operate numerous short-lived objects indexed in hash tables or trees [32, 69, 82, 90, 122], e.g. for grouping events by key. They manage these objects with generic memory allocators [82] or garbage collectors [87, 122]. Such designs poorly fit a TEE's small TCB and limited DRAM portion, e.g. typically tens of MB for TrustZone TEE and up to 128 MB for Intel SGX enclave [31]. iii) ***Verifying stream analytics results***. This requires to track unbounded data flows in stream pipelines, validate if operators respect the temporal properties, e.g. windows, and minimize the resultant overhead in execution and communication.

Why are existing systems inadequate? First, many TEE-based systems [22, 27, 112] pull entire user applications and libraries to the TCB, as shown in Figure 4(a). However, as we described in Section 2.2, a modern analytics engine and its libraries are large, complex, and potentially vulnerable. Second, partitioning applications to suit a TEE, as shown in Figure 4(b) [71, 93, 101], is unsuitable for existing stream engines: partitioning does not change their hash-based data structures and algorithms, which by design mismatch a TEE. Similarly, recent secure processing engines disfavor partitioning [89, 91]. Third, recent systems use TEE to protect data in analytics or in network packet processing. As summarized in Table 1, they lack support for stream analytics, key computation optimizations, or specialized memory allocation, which we will demonstrate as vital to our objective.

Attesting TEE integrity [65, 91] is insufficient to assert analytics correctness. VC3 [99] and Opaque [124] verify correctness of *batch* analytics by checking the history of compute results, i.e. their data lineage [50, 102]. Without tracking data being continuously ingested and lacking a stream model, data lineages cannot assert whether *all* ingested data is processed according to pipeline declarations, watermarks, and temporal windows, which are critical to stream analytics.

4.2 StreamBox-TZ in a Nutshell

SBT builds on TrustZone [2] due to ARM's popularity for the edge and trusted IO benefiting stream analytics (§2).

Programmability Programming SBT is similar to programming commodity engines such as Spark Streaming [122] and Flink [19]. Analytics programmers assemble pipelines with high-level, declarative operators as exemplified in Figure 2(c). SBT provides most of the common operators offered by commodity engines, as summarized in Table 2. These stream operators are widely used for analytics over telemetry data [62, 90]. SBT supports User Defined Functions (UDFs) that are certified by a trusted party, which is a common requirement in TEE-based systems [91].

SBT architecture As shown in Figure 3, SBT's data plane incarnates as a TrustZone module. SBT runs its control plane as a parallel runtime in the normal world. The control plane invokes the data plane through a narrow interface (details in Section 9). The control plane orchestrates the execution of analytics pipelines. It creates plentiful parallelism among and within operators. It elastically maps the parallelism to a

pool of threads it maintains. At a given moment, all threads may simultaneously execute one operator as well as different operators over different data.

Data plane & design choices SBT's data plane consists of only the trusted primitives and a runtime for them.

i) Trusted primitives are stateless, single-threaded functions that are oblivious to synchronization. We do not enclose whole stream pipelines in the data plane, because a stream pipeline must be scheduled dynamically for parallelism and handling high-velocity data. We do not enclose whole declarative operators in the data plane, because one operator instance has internal thread-level parallelism and hence requires thread management logic. Our choice keeps the data plane lean, leaving out all control functions including scheduling and threading. This contrasts to many other engines pulling whole analytics to TEE as shown in Table 1.

Although exporting low-level primitives entails more TEE switches, the costs are lower on modern ARM [25, 56] and can be amortized by data batching, as will be discussed soon.

ii) The data plane incorporates minimum runtime functions: memory management and paging, which are critical to TEE integrity; cache coherence of parallel primitives, which is critical to parallelism. The data plane is agnostic to declarative operators and pipelines being executed.

For attestation, the data plane generates audit records on data ingress/egress, watermarks, and primitive executions. It reduces overhead via data batching and record compression.

Coping with secure memory shortage When compute cost or data ingestion rate is high, SBT may run short of secure memory. To avoid data loss in such a situation, SBT adds backpressure to source sensors, slowing down data ingestion. In the current implementation, SBT triggers backpressure when ingestion exceeds a user-defined threshold; we leave as future work automatic flow control, i.e. tuning the threshold online per available secure memory and backlog.

5 Trusted Primitives and Optimizations

Parallel execution inside a TEE SBT exploits task parallelism without bloating the TEE with a threading library. The control plane invokes multiple primitives from multiple worker threads, which then enter the TEE to execute the primitives in parallel. All trusted primitives share one cache-coherent memory address space in TEE, which simplifies data sharing and avoids copy cost. This contrasts to existing secure analytics engines that leave task parallelism untapped in a single TEE [53, 99].

Array-based algorithms to suit TEE Unlike many popular stream engines using hash-based algorithms for lower algorithmic complexity, we make a new design decision. We strongly favor algorithms with simple logic and low memory overhead, despite that they may incur higher algorithmic complexity. Corresponding to contiguous arrays as the universal data containers in TEE, most primitives use sequential-access algorithms over contiguous arrays, e.g. executing Merge-Sort over event arrays and scanning the resultant array to calculate the average value per key.

Trusted primitives and vectorization SBT's trusted primitives are generic. They constitute most declarative stream operators, often referred to as Select-Projection-Join-GroupBy (SPJG) families, shown in Table 2. These operators are considered representative in prior research [44].

To speed up the array-based algorithms inside TEE without TCB bloat, our insight is to map their internal data parallelism to vector instructions of ARM [21]. Despite their well-known performance benefit, vector instructions are rarely used to accelerate data analytics *within* TEEs, to our knowledge. Vectorization incurs low code complexity as the performance gain comes from a CPU feature that is already part of the TCB.

Our optimization focuses on Sort and Merge, two core primitives that dominate the execution of stream analytics according to our observation. Inspired by vectorized sort and merge on x86 [26, 64], we build new implementations for SBT by hand-writing ARMv8 NEON vector instructions. Our sort outperforms the ones in the C/C++ standard libraries by more than 2×, as will be shown in evaluation. This optimization is crucial to the overall engine performance.

6 TEE Memory Management

Facing high-velocity streams in a TEE, SBT's memory allocator addresses two challenges: *space efficiency*: it must create compact memory layout and reclaim memory timely due to limited physical memory; *lightweight*: the allocator must be simple to suit a low TCB. The challenges disqualify popular engines that organize events in hash tables (e.g. for grouping events by key) and rely on generic memory allocators [32, 69, 82, 90, 122]. The reasons are two: a hash table's principle of trading space for time mismatches TEE's limited memory; generic allocators often feature sophisticated optimizations, adding tens of KSLoC to TCB [41, 59].

SBT specializes memory management for stream computations: it supports unbounded buffers as the universal memory abstraction (§6.1); it places data by using (untrusted) consumption hints and large virtual address space (§6.2).

6.1 Unbounded Array

We devise contiguous, virtually unbounded arrays called *uArrays*, a new abstraction as the universal data containers used by computations in TEE. uArrays encapsulate all the data in a pipeline, including data flowing among trusted primitives as well as operator states traditionally kept in hash tables.

An uArray is an append-only buffer in a contiguous memory region for same-type data objects. Their lifecycles closely

map to the producer/consumer pattern in streaming computations. One uArray can be in three states. *Open*: after an uArray is created, it dynamically grows as the producer primitive appends data objects to it. *Produced*: the data production completes and the end position of the uArray is finalized. uArray becomes read-only and no data can be appended. *Retired*: the uArray is no longer needed and its memory is subject to reclamation. The memory allocator places and reclaims uArrays regarding their states, as will be discussed in Section 6.2.

Types uArrays fall into different types depending on their scopes and enclosed data. A *streaming uArray* encapsulates data flowing from a producer primitive to a consumer primitive. A *state uArray* encapsulates operator state that outlives the lifespans of individual primitives. A *temporary uArray* live within a trusted primitive's scope.

Low abstraction overhead An uArray spans a contiguous virtual memory region and grows transparently. The growth is backed by the data plane's on-demand paging that completely happens in the TEE. For most of the time, growing an uArray only requires updating an integer index. Compared to manually managed buffers, this mechanism waives bounds checking of uArray in computation code and hence allows the compiler to generate more compact loops. uArrays always grow in place. This contrasts to common sequence containers (e.g. C++ `std::vector` and `java.util.ArrayList`) that grow transparently but require expensive relocation. We will experimentally compare uArray with `std::vector` in Section 9.

6.2 Placing uArrays in uGroups

Co-locating uArrays The memory allocator co-locates multiple uArrays as a uGroup in order to reclaim them consecutively. Spanning a contiguous virtual memory region, a uGroup consists of multiple *produced* or *retired* uArrays and optionally an *open* uArray at its end, as shown in Figure 5. The grouping is purely physical: it is at the discretion of the allocator, orthogonal to stream computations, and therefore transparent to the trusted primitives and the control plane.

With the grouping, the allocator reclaims *consumed* uArrays by always starting from the beginning of an uGroup, as shown in Figure 5. To place a new uArray, the allocator decides whether to create a new uGroup for the uArray, or append the uArray to an existing uGroup. In doing so, the allocator seeks to i) ensure that each uGroup holds a sequence of uArrays to be consumed consecutively in the future; ii) minimize the total number of live uGroups, in order to compact TEE memory layout and minimizes the cost in tracking uGroups. To this end, our key is to guide placement with the control plane's data consumption plan, as will be presented below.

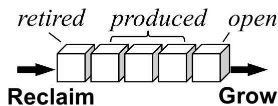


Figure 5: The uArrays in one uGroup

Consumption hints Upon invoking a trusted primitive T, the control plane may provide two optional hints concerning the future consumption order for the output of T:

- *Consumed-in-parallel* ($\|_k$): the control plane will schedule k worker threads to consume a set of uArrays in parallel.
- *Consumed-after* ($b_1 \Leftarrow b_2$): the control plane will schedule worker threads for consuming uArray b_2 after uArray b_1. The *consumed-after* relation is transitive. uArrays may form multiple *consumed-after* chains.

The control plane may specify these relations between new output uArrays (yet to be created) and existing uArrays.

Hint-guided placement The hints assist the data plane to generate compact memory layout and reclaim memory effectively. Upon allocating a uArray, the allocator examines the existing hints regarding to the uArray.

($\Leftarrow$) prompts the allocator to place the uArrays on the same *consumed-after* chain in the same uGroup. Starting from the new uArray b under question, the allocator tracks back on its consumed-after chain, and places b after the first uArray that is both in state *produced* (i.e. its growth has finished) and is located at the end of an uGroup. If no such uArray is available on the chain, the allocator creates a new uGroup for b.

($\|_k$) prompts the allocator to place uArrays $b_{1..k}$ in separate uGroups, so that delay in consuming any of the uArrays will not block the allocator from reclaiming the other uArrays. Our rationale is that despite $b_{1..k}$ are created at the same time, they are often consumed at different moments in the future: i) since SBT's control plane threads independently fetch new uArrays for processing as they become available (§4), the starting moments for processing $b_{1..k}$ may vary widely, especially when the engine load is high; ii) even when k worker threads start processing $b_{1..k}$ simultaneously, straggling workers are not uncommon, due to non-determinism of a modern multicore's thread scheduling and memory hierarchy [24].

The impacts of misleading hints SBT detects misleading hints in retrospect through remote attestation (§7). As the hints only affect TEE memory placement *policy* on the edge, misleading hints never result in data loss (§4.2) or violation of data security and TEE integrity. Yet, such hints may slow down analytics and therefore violate result freshness.

Managing virtual addresses All uGroups grow *in place* within one virtual address space. To avoid collision and expensive relocation, the allocator places them far apart by leveraging the large virtual address space dedicated to a TrustZone TEE. The space is 256TB on ARMv8, 10,000× larger than the physical DRAM (a few GBs). Hence, the allocator simply reserves for each uGroup a virtual address range as large as the total TEE DRAM. We will validate this choice in Section 9.

7 Attestation for Correctness and Freshness

SBT collects evidences for cloud consumers to verify two properties: *correctness*, i.e. all ingested data is processed ac-

Field	Description	Length
Ts	Data plane timestamp	32 bits
Op	Primitive type, including ingress/egress	16 bits
WinNo	Monotonic window sequence number	16 bits
Data	An uArray ID or a watermark value	32 bits
Hint	An optional consumption hint	64 bits
Count	Number of data/hint fields that follow	16 bits

In/Egress	Op	Ts	Data					
Windowing	Op	Ts	Data	WinNo	Data			
Execution	Op	Ts	Cnt	Data...	Cnt	Data...	Cnt	Hints...

Figure 6: Audit records: fields (top) and layout (bottom)

cording to the stream pipeline declaration; *freshness*, i.e. the pipeline has low output delays.

The above objective has several notable aspects. i) We verify the behaviors of untrusted control plane, i.e., *which* primitives it invokes on *what* data and at *what* time. We do not verify trusted primitives, e.g. if a Sort primitive indeed produces ordered data. ii) Verifying data lineages at the pipeline's intermediate operators or egress [50, 102] is insufficient to guarantee correctness, i.e. all data ingested so far is processed according to the stream pipeline. iii) The windows of stream computations and watermarks triggering the computations must be attested, which are keys to stream model (§2). iv) As the volume of evidences can be substantial, evidences must be compacted to save uplink bandwidth [84, 114].

Therefore, SBT provides the following verification mechanism. Agnostic to the pipeline being executed, the data plane monitors dataflows among primitive instances at the TEE boundary, and then generates audit records. For low overhead, it eschews building data lineages on-the-fly unlike much prior work [50,74,99]. The data plane compresses audit records and flushes to the cloud both periodically and upon externalizing any analytics result. We describe details below.

Audit records As being invoked by the control plane, the data plane generates *audit records*. As illustrated in Figure 6, the records track i) ingested and externalized uArrays, ii) associations between uArrays and windows, and iii) primitive executions (with optional hints supplied by the control plane) which establish *derived-from* relations among uArrays. The records further include ingested watermark values, which are crucial for determining output delays as will be discussed below. The data plane timestamps all the records. It generates monotonically increasing identifiers for recorded uArrays. We will evaluate the overhead of audit records in Section 9.

Attesting analytics correctness The cloud verifier checks if all ingested uArrays flow through the expected trusted primitives. Such dataflows are deterministic given the arrivals of input data (including their windows), the watermarks, and the pipeline declaration. Hence, the verifier replays all ingestion records on its local copy of the same pipeline. It checks if all the records resulting from the replay match the ones reported by the edge (except timestamps). The replay is symbolic without actual computations and hence fast.

Note that the verification works for stateful operators as well. The state of a stream operator (e.g. temporal join) is only determined by all the inputs the operator has ever received. Since the cloud can verify that all the ingested uArrays correctly flow through the expected trusted primitives and thus stream operators, it knows that the operator's current state must be correct, and then all results derived from the operator state must be correct.

Attesting result freshness The key for the verifier to calculate the delay of an output result R is to identify the watermark that triggers the externalization of R, according to the delay definition in Section 2.2. From the egress record of R, the verifier traces *backward* following the *derived-from* chain(s) until it reaches an execution record indicating that a watermark W triggers the execution. The verifier looks up the ingress record of W. It calculates the difference between W's ingress time and R's egress time to be the delay of R.

Example In Listing 1, an uArray with identifier 0xF0 is ingested and segmented into two uArrays (0xF1 and 0xF2) for window 0 and 1 respectively. Sort consumes uArray 0xF1 and produces uArray 0xF3. A watermark with value 100 arrives and completes window 0. Triggered by the watermark, SUM consumes uArray 0xF3 of window 0 and produces uArray 0xF5 as the result of window 0.

```
ts= 1  INGRESS data=0xF0
ts= 5  WND data_in=0xF0 win_no=0 data_out=0xF1
ts=10  SORT data_in=0xF1 data_out=0xF3
ts=15  INGRESS data=0xF4 (watermark=100)
ts=25  SUM data_in=0xF3,0xF4 data_out=0xF5
ts=28  WND data_in=0xF0 win_no=1 data_out=0xF6
ts=30  EGRESS data=0xF5
```

Listing 1: Sample audit records for the pipeline in Figure 2. Format is simplified. ts means processing timestamp.

The cloud verifier replays the ingress records on its local pipeline copy and learns that uArray 0xF1 is processed adhering to the pipeline declaration while uArray 0xF2 is yet to be processed. It will assert analytics incorrectness if 0xF2 remains unprocessed until a future watermark completes window 1 (not shown). To verify result freshness, the verifier traces result 0xF5 backward to find its trigger watermark 0xF4 and calculates the output delay to be 15 ($30-15$).

Columnar compression of records The data plane compresses audit records by exploiting locality within one record field and known data distribution in each field. The data plane produces raw audit records in memory (with the format shown in Figure 6) and in a row order, i.e. one record after the other. Before uploading a sequence of records, it separates the record fields (i.e. columns) and applies different encoding schemes to individual columns: i) Huffman encoding for primitive types and data counts, the two columns likely contain skewed values; ii) delta encoding for timestamps, uArray identifiers, and window numbers, which increment monotonically. Our compression is inspired by columnar databases [107]. We will evaluate the efficacy of compression in Section 9.

8 Implementation

We build SBT for ARMv8 and atop OP-TEE [70] (v2.3). SBT reuses most control functions of StreamBox [82], an open-source research stream engine for x86 servers. Yet, as StreamBox mismatches a TEE (§4.1), SBT contributes a new architecture and a new data plane. SBT communicates with source sensors and cloud consumers over ZeroMQ TCP transport [57] which is known for good performance. The new implementation of SBT includes 12.4K SLoC.

Input batch size, a key parameter of SBT, trades off between delays in executing individual primitives, the rate of TEE entry/exit, and attestation cost. We empirically determine it as 100K events and will evaluate its impact (§9). **Opaque references** for uArrays are 64-bit random integers generated by the data plane. It keeps the mappings from references to uArray addresses in a table, and validates opaque references by table lookup. This incurs minor overhead, as live opaque references are often no more than a few thousands.

9 Evaluation

We answer the following questions through evaluation:

- Does SBT result in a small TCB? (§9.1)
- What is SBT's performance and how is it compared to other engines? What is the overhead? (§9.2)
- How do our key designs impact performance (§9.3)?

9.1 TCB Analysis

TCB size Table 4 shows a breakdown of the SBT source code. Despite a sophisticated control plane, the data plane only adds 5K SLoC to the TCB. SBT's memory management is in 740 SLoC, 9× fewer than glibc's `malloc` and 20× fewer than `jemalloc` [41]. The size of data plane is 42.5 KB, a small fraction (16%) of the entire OP-TEE binary.

TCB interface The SBT's data plane exports only four entry functions: two for data plane initialization/finalization, one for debugging, and one shared by all 23 trusted primitives. The last function accepts and returns opaque references (§4). No state is shared across the protection boundary.

Comparison with alternative TCBs Compared to enclosing whole applications in TCB [22, 27, 112], SBT keeps most of the engine out, shrinking the TCB by at least one order of magnitude. Compared to directly carving out [71, 93] the original StreamBox's data functions for protection, SBT completely avoids sophisticated data structures (e.g. AtomicHashMap [47] used by StreamBox) that mismatch TCB. Compared to VC3 [99] that implements Map/Reduce operators in a TCB with ∼9K SLoC, SBT supports much richer stream operators within a 2× smaller TCB.

SoC	HiSilicon Kirin 620, TDP 36W	**CPU**	8x ARM Cortex-A53@1.2 GHz
Mem	2GB LPDDR3@800 MHz	**OS**	Normal: Debian 8 (Linux 4.4) Secure: OP-TEE 2.3

Table 3: The test platform used in experiments

Data Plane (Trusted)

Primitives*	Mem Mgmt*	Misc*	**Total**
3.7K (32.5 KB)	0.7K (6 KB)	0.6K (4 KB)	**5K (42.5 KB)**

Control Plane (Untrusted)

Control	Data types*	Operators*	Test*	Misc*	**Total**
23K	1.3K	4.1K	1K	1K	**31K (348 KB)**

Major Libraries (Untrusted)

glibc 2.19	libstdc++ 3.4.2	libzmq 2.2	boost 1.54	**Total**
1135K	110K	13K	37K	**1.3M (3.1 MB)**

* New implementations of this work. Total = 12.4K SLoC.

Table 4: A breakdown of the StreamBox-TZ source, of which 5K SLoC are in TCB. Binary code sizes shown in parentheses

9.2 Performance & Overhead

Methodology We evaluate SBT on a HiKey board as summarized in Table 3. We chose HiKey for its good OP-TEE support [70] and that it is among the few boards with TrustZone programmable by third parties. We built *Generator*, a program sends data streams over ZeroMQ TCP transport [57] to SBT. We run the cloud consumer on an x86 machine. Data streams are encrypted with 128-bit AES.

In the face of HiKey's platform limitations, we set up the engine ingestion as follows. i) Although Gigabit Ethernet on edge platforms is common [5, 88], Hikey's Ethernet interface (over USB) only has 20MB/sec bandwidth. We have verified that the interface is saturated by SBT with 4 cores. Hence, we report performance when SBT and *Generator* both run on HiKey communicating over ZeroMQ TCP, which still fully exercise the TCP/IP stack and data copy. ii) Although HiKey's TEE is capable of directly operating Ethernet interface as trusted IO, our OP-TEE version lacks the needed drivers. Hence, we emulate SBT's direct data ingestion to TEE by running the ingestion in a privileged process in the normal world, and bypassing data copy across the TEE boundary. Our test harness continuously replays pre-allocated secure memory buffers populated with events.

As summarized in Table 5, we test SBT as well as three modified versions: ***SBT ClearIngress*** ingests data in cleartext; this is allowed if source-edge links are trusted as defined in our threat model (§3). ***SBT IOviaOS*** does not exploit TrustZone's trusted IO: the untrusted OS ingested (encrypted) data and copies the data across TEE boundary to the data plane. ***Insecure*** completely runs in the normal world with ingress and egress in cleartext, showing native performance. This is basically StreamBox [82] with SBT's optimized stream computations (§5). We report the engine performance as its maximum input throughput when the pipeline output delay (defined in §2.2) remains under a target set by us.

Benchmarks We employ six benchmarks of processing sensor data streams from prior work [32, 62, 63, 67, 82]. They

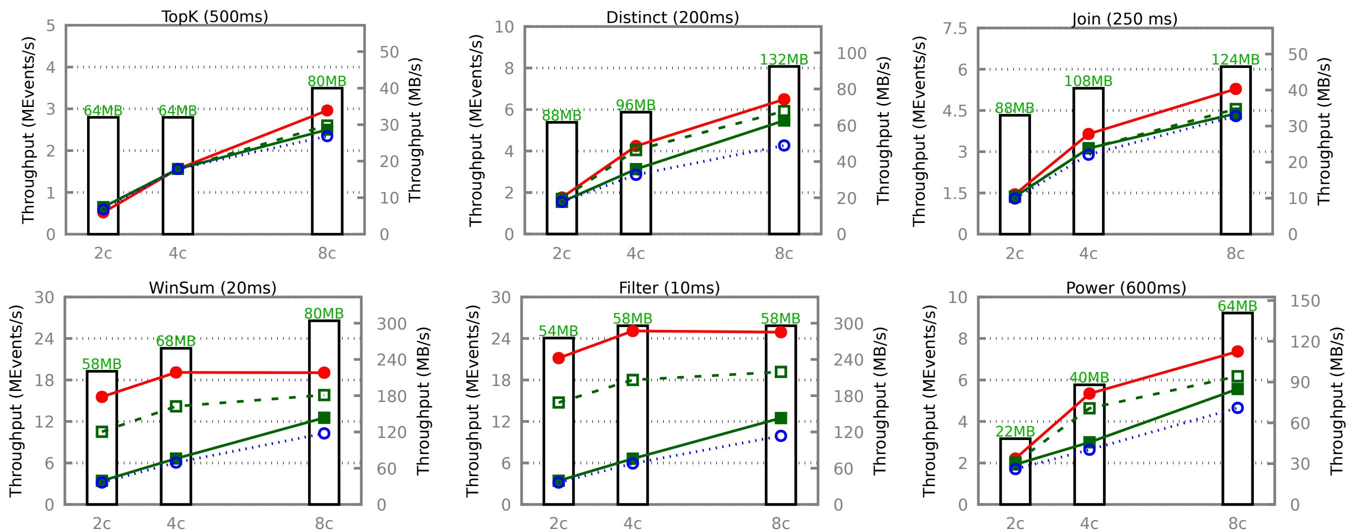


Figure 7: StreamBox-TZ throughput (lines, left/right y-axes) as a function of CPU cores (x-axis) under given output delays (above each plot). Steady consumptions of TEE memory as columns with annotated values. See Table 5 for legends and explanations.

Legend & Version	Data Plane	In/Egress Path	Ingress Data	Egress Data
StreamBox-TZ	in TEE	Trusted IO*	Encrypted	Encrypted
SBT ClearIngress	in TEE	Trusted IO*	ClearTxt	Encrypted
SBT IOviaOS	in TEE	via OS	Encrypted	Encrypted
Insecure#	out TEE	in OS	ClearTxt	ClearTxt

* Through TrustZone Trusted IO directly to TEE
\# Equivalent to a StreamBox invoking StreamBox-TZ's optimized stream compute

Table 5: Engine versions for comparison (plots in Figure 7)

cover major stream operators and a variety of pipelines. We use fixed windows, each encompassing 1M events and spanning 1 second of event time. Each event consists of 3 fields (12 Bytes) unless stated otherwise. (1) **Top Values Per Key (TopK)** groups events based on keys and identifies the K largest values in each group in each window. (2) **Counting Unique Taxis (Distinct)** identifies unique taxi IDs and counts them per window. For input events, we use a dataset of taxi trip information containing 11 K distinct taxi IDs [63]. (3) **Temporal Join (Join)** joins events that have the same keys and fall into same windows from two input streams. (4) **Windowed Aggregation (WinSum)** aggregates input values within each window. We use the Intel Lab Data [75] consisting of real sensor values as input. (5) **Filtering (Filter)** filters out input data, of which field falls into to a given range in each window. We set 1% selectivity as done in prior work [67]. (6) **Power Grid (Power)**, derived from a public challenge [62], finds out houses with most high-power plugs. Ingesting a stream of per-plug power samples, it calculates the average power of each plug in a window and the average power over all plugs in all houses in the window. For each house, it counts the number of plugs that have a higher load than average. It emits the houses that have most high-power plugs in the window. The event for this benchmark is composed of 4 fields (16 Bytes).

Benchmark 2, 4, and 6 use real-world datasets; others use synthetic data sets of which fields are 32-bit random integers. Note that SBT's `GroupBy` operator bases on sort and merge and is insensitive to key skewness [15].

End-to-end performance Figure 7 shows the throughputs of all benchmarks as a function of hardware parallelism. SBT can process up to multiple millions of events within sub-second output delays (labeled atop each plot). For simpler pipelines such as WinSum and Filter, SBT processes around 12M events/sec (140 MB/sec). This throughput saturates one GbE link which is common on IoT gateways [88]. Overall, SBT can use all 8 cores in a scalable manner.

SBT's absolute performance is state of the art. We test three popular, insecure stream engines: Flink [19], designed for distributed environment and known for good single-node performance [68]; Esper [46], designed for a single machine; SensorBee [90], designed for sensor data processing on a single device. As shown in Figure 8, on the same hardware (HiKey) and the same benchmark (WinSum), we have measured that SBT's throughput is at least one order of magnitude higher than the others. This is because i) our *Insecure* baseline has high performance for its rich task parallelism (inherited from StreamBox [82]) and native, vectorized stream computations (new contributions); ii) SBT only imposes modest security overhead, as will be shown later.

Comparison to secure stream engines The comparison is challenged by that TrustZone was rarely exploited for protecting data-intensive computations. To our knowledge, i) no analytics engines use TrustZone for data protection and ii) no systems can partition an insecure stream engine for TrustZone. Note that popular secure analytics engines, e.g. VC3 [99] and Opaque [124], not only require SGX but also target batch processing instead of stream analytics. To this end, we qualitatively compare SBT with SecureStreams [53], the closest system we are aware of. Designed for an x86 cluster, SecureStreams uses SGX to protect stream operators and targets strong data security. On a benchmark similar to WinSum it

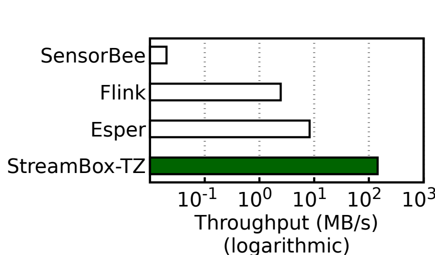


Figure 8: StreamBox-TZ achieves much higher throughput than commodity insecure engines [19, 46, 90] on HiKey. Benchmark: windowed aggregation; target output delay: 50ms.

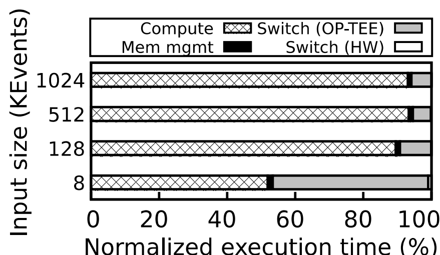


Figure 9: Run time breakdown of operator `GroupBy` under different input batch sizes. The control plane runs 8 threads to execute `GroupBy` in parallel. Total execution time is normalized to 100%.

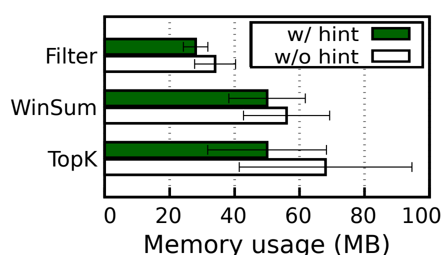


Figure 10: Without consumption hints, the allocator uses more TEE memory. Since memory usage fluctuates at run time, the error bars show two standard deviations below and above the average.

was reported to achieve 10 MB/sec, one magnitude lower than SBT on WinSum. Furthermore, SecureStreams achieved such performance on a small x86 cluster which has much richer resource than HiKey: the former has faster CPUs (8x i7-6700@3.4GHz versus 8x Cortex-A53@1.2GHz), larger DRAM (16 GB versus 2 GB), higher power (130W versus 36W), and higher cost ($600 versus $65).

SBT's advantage comes from i) data exchange via coherent memory inside one TEE, instead of exchanging encrypted messages among workers; ii) memory management specialized for streaming, and iii) vectorized computations.

Security overhead We investigate the overhead of the new security mechanism contributed by SBT – its isolated data plane. We assess the overhead as the throughput loss of *SBT ClearIngress* as compared to *Insecure* (i.e. native performance as StreamBox [82] invoking SBT's stream computations), both paying same costs for data ingress. The target output delays are the same (labeled atop each plot in Figure 7). The security overhead is less than 25% in all benchmarks. This is similar to or lower than the reported overhead (20–70%) in recent TEE systems [22, 71, 112]. ***Overhead analysis:*** The security overhead mostly comes from world switch, among operators and inside each operator. To understand the switch cost within an operator, we profile `GroupBy`, one of the most costly operators. We test different input batch sizes, which have a strong impact on TEE entry/exit rates and hence isolation overhead (§4). Figure 9 shows a run time breakdown. When each input batch contains 128K (close to the value we set for SBT) or more events, more than 90% of the CPU time is spent on actual computations in TEE. The CPU usage of TEE memory management is as low as 1–2%. In the extreme case where each input batch contains as few as 8K events, the overhead of world switch starts to dominate. Most of the world switch overhead comes from OP-TEE instead of the CPU hardware (a few thousand cycles per switch), suggesting room for OP-TEE optimization.

Impact of decrypting ingress data Decrypting ingress data is needed if source-edge links are untrusted (§3) and source must send encrypted data. It has substantial performance impact. By comparing SBT to *SBT ClearIngress*, turning on/off ingress decryption leads to 4% – 35% throughput difference when all 8 cores are in use. The performance gap is more pronounced for simple pipelines, which has higher ingestion throughput leading to higher decryption cost.

TEE memory usage While sustaining high throughput, SBT consumes a moderate amount of physical memory, ranging from 20 MB to 130 MB as shown in Figure 7. The memory usage is as low as 1–6% of the total system DRAM. The virtual memory usage is also low, often 1–5% of the entire virtual address space in OP-TEE. The memory usage increases with the throughput, since there will be more in-flight data. On the same platform, Flink's memory consumption is 3× higher, due to its hash-based data structures and the use of JVM. This validates our choice of uArrays.

Attestation overhead Attestation incurs minor overhead to both the edge and the cloud. We measured that SBT produces 300–400 audit records per second across all our benchmarks, and spends a few hundred cycles on producing each record. Compressing such record streams on HiKey consumes 0.2% of total CPU time. Our consumer written in Python on a 4-core i7-4790 machine replays 57K records per second with a single core, suggesting a capability of attesting near 500 SBT instances simultaneously. We will evaluate the efficacy of record compression in Section 9.3.

9.3 Validation of Key Design Features

Exploitation of trusted IO As shown in Figure 7, a comparison between SBT and *SBT IOviaOS* demonstrates the advantage of directly ingesting data into TEE and bypassing the OS: SBT outperforms the latter by up to 20% in throughput due to reduction in moving ingested data.

Trusted primitive vectorization (§5) Our optimizations with ARM vector instructions are crucial. To show this, we examine `GroupBy`, one of the top hotspot operators. When we replace the vectorized Sort that underpins `GroupBy` with two popular implementations (qsort() from the the OP-TEE's libc and std::sort() from the standard C++ library), we measured the throughput of `GroupBy` drops by up to 7× and 2×,

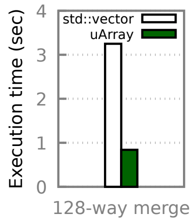
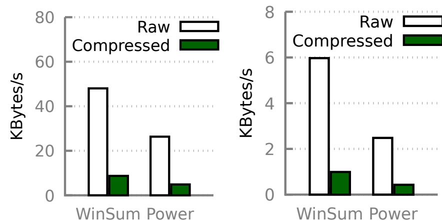


Figure 11: On-demand growth of uArrays vs. std::vector



(a) 10K events/batch (b) 100K events/batch

Figure 12: Compression of audit records saves uplink bandwidth substantially.

respectively. We have similar observation on other operators.

Efficacy of hint-guided memory placement (§6.2) We compare to an alternative design: the modified allocator acts based on the heuristics that all the uArrays produced by the same primitive belong to the same *generation* and are likely to be reclaimed altogether. Accordingly, the modified allocator places these uArrays in the same uGroup. As shown in Figure 10, in three benchmarks, the modified allocator increases memory usage by up to 35%. This is because, without hints, it cannot place uArrays based on future consumption.

uArray on-demand growth (§6.1) We compare uArray to std::vector, a widely used C++ sequence container with on-demand growth. We run a microbenchmark of N-way merge, an intensive procedure in trusted primitives. It iteratively merges 128 buffers (uArrays or vectors), each containing 512 KB (128K 32-bit random integers) until obtaining a monolithic buffer; as merge proceeds, buffers grow dynamically. As shown in Figure 11, uArrays is 4× faster than std::vector, because the allocation and paging in TEE that back uArray growth is much faster than that of a commodity OS.

Compression of audit records (§7) The compression significantly saves the uplink bandwidth. We test two benchmarks (WinSum and Power) on two extremes of the spectrum of computation cost, and test two very different input batch sizes. This is because simpler computations and smaller batch sizes generate audit records at higher rates. Figure 12 shows that SBT compresses audit records by 5×–6.7×. In an offline test using gzip to compress the same records, we find our compression ratios are 1.9× higher than gzip. 2–40 KB/sec of uplink bandwidth is saved, which is significant compared to the uploaded analytics results, which are 144 bytes/sec for WinSum and 400 bytes/sec for Power.

10 Related Work

Secure data analytics DARKLY [61] protects sensor data by isolating computations in an OS process, resulting in a large TCB. VC3 [99] and SecureStreams [53] use SGX to protect the operators in distributed analytics. They lack optimizations for parallel execution in one TEE on the edge. To process data confidentiality, STYX [106] computes over encrypted data, a method likely prohibitively expensive to edge platforms. Opaque [124] protects data access patterns of distributed operators, targeting a threat out of our scope.

TCB minimization Minimizing TCB is a proven approach towards a trustworthy system. Flicker [80] directly executes security-sensitive code on baremetal hardware. Trustvisor [79] shrinks its TCB to a specialized hypervisor. Sharing a similar goal, SBT addresses unique challenges in supporting data-intensive computation on a minimal TCB.

Trusted Execution Environments Much work isolates security-sensitive software components. Terra [48] supports isolation with a virtual machine. Many systems used TrustZone and SGX [81] for TEE. Some systems enclose in TEE whole applications [22, 27, 51, 112], while others partition existing programs for TEE [71, 93, 101]. These approaches often result in larger TCBs and/or higher overhead than SBT and are thus less desirable for SBT. TEE also sees various novel usage, including protecting mobile app classes [96], enforcing security policies [30], remote attestation of application control flows [13], and controlling data access [34]. None addresses data-intensive computation as SBT does.

Edge processing evolves from a vision [98, 100] to practice [37, 42, 83]. Most works focused on programming paradigms [94], developing and deploying application [29, 52, 114], and resource management [86]. Complementary to them, SBT focuses on secure analytics on the edge.

Stream processing systems, in response to big data challenges, evolve from single-threaded [33, 40, 78, 105, 110] to massive parallel systems [14, 69, 85, 92, 92, 113, 122]. The existing systems focus on challenges, such as fault tolerance [122], fast reconfiguration [115], high parallelism [32, 82], and the use of GPUs [67]. Few systems achieve data security and performance simultaneously as SBT does.

11 Conclusions

This paper presents StreamBox-TZ (SBT), a secure stream analytics engine designed and optimized for a TEE on an edge platform. SBT offers strong data security, verifiable results, and competitive performance. On an octa core ARM machine, SBT processes up to tens of millions of events per second; its security mechanisms incur less than 25% overhead.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd, Eyal de Lara, for their insightful comments. For this project: the authors affiliated with Purdue ECE were supported in part by NSF Award #1718702, #1619075, Purdue University CPS/IoT Seed Grant Program, and a Google Faculty Award; the author affiliated with Northeastern University was supported in part by NSF Award #1748334.

References

[1] Apache Beam. https://beam.apache.org/.

[2] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[3] CVE-2010-3190: Untrusted search path vulnerability in the microsoft foundation class (mfc) library. https://nvd.nist.gov/vuln/detail/CVE-2010-3190.

[4] CVE-2017-12629: Remote code execution occurs in apache solr. https://nvd.nist.gov/vuln/detail/CVE-2017-12629.

[5] Marvell Armada 8K family processors. http://www.marvell.com/embedded-processors/armada-80xx/.

[6] CVE-2008-0171: Boost.regex allows context-dependent attackers to cause failed assertion and crash. https://nvd.nist.gov/vuln/detail/CVE-2008-0171s, 2008.

[7] CVE-2009-2493: Active template library does not properly restrict use of oleloadfromstream in instantiating objects from data streams, which allows remote attackers to execute arbitrary code. https://nvd.nist.gov/vuln/detail/CVE-2009-2493, 2009.

[8] CVE-2015-4421: in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4421, 2015.

[9] CVE-2015-4422: in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4422, 2015.

[10] CVE-2016-10229: udp.c in the linux kernel before 4.5 allows remote attackers to execute arbitrary code. https://nvd.nist.gov/vuln/detail/CVE-2016-10229, 2016.

[11] CVE-2017-11176: The mq_notify function in the linux kernel allows attackers to cause a denial of service or possibly have unspecified other impact. https://nvd.nist.gov/vuln/detail/CVE-2017-11176, 2017.

[12] CVE-2018-8822: Incorrect buffer length handling in the ncp_read_kernel function could be exploited by malicious ncpfs servers to crash the kernel or execute code. https://nvd.nist.gov/vuln/detail/CVE-2018-8822, 2017.

[13] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endow.*, 8(12):1792–1803, 2015.

[15] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endow.*, 5(10):1064–1075, 2012.

[16] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[17] M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler. Distil: Design and implementation of a scalable synchrophasor data processing system. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2015.

[18] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, 1997.

[19] Apache. Apache flink: Scalable stream and batch data processing. https://flink.apache.org/, 2017.

[20] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *Proceedings of the VLDB Journal*, 15(2):121–142, 2006.

[21] Arm. Arm neon technology. https://developer.arm.com/technologies/neon, 2018.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] Art Manion. CERT/CC Blog – Anatomy of Java Exploits. https://insights.sei.cmu.edu/cert/2013/01/anatomy-of-java-exploits.html/, 2013.

[24] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Proceedings of Commun. of the ACM*, 55(5):111–119, 2012.

[25] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[26] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endow.*, 7(1):85–96, 2013.

[27] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[28] A. Becher, Z. Benenson, and M. Dornseif. Tampering with motes: Real-world physical attacks on wireless sensor networks. In *Proceedings of the 3rd International Conference on Security in Pervasive Computing*, 2006.

[29] K. Bhardwaj, M. W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[30] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[31] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, 2016.

[32] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endow.*, 8(4):401–412, 2014.

[33] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[34] F. Chen. *Cross-platform data integrity and confidentiality with graduated access control*. PhD thesis, The University of British Columbia, 2016.

[35] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, 2011.

[36] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. 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 (ASPLOS)*, 2008.

[37] CISCO. White paper: The cisco edge analytics fabric system. http://www.cisco.com/c/dam/en/us/products/collateral/analytics-automation-software/edge-analytics-fabric/eaf-whitepaper.pdf, 2016.

[38] R. Clapis. Go get my/vulnerabilities: an in-depth analysis of go language. https://www.blackhat.com/docs/asia-17/materials/asia-17-Clapis-Go-Get-My-Vulnerabilities-An-In-Depth-Analysis-Of-Go-Language-Runtime-And-The-New-Class-Of-Vulnerabilities-It-Introduces.pdf, Blackhat Asia 2017.

[39] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[40] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[41] J. E. David Goldblatt, Dave Watson. Jemalloc memory allocator. http://http://jemalloc.net/, 2017.

[42] Dell. Dell further democratizes advanced analytics with latest release of statistica. http://www.dell.com/learn/us/en/uscorp1/press-releases/2016-04-14-dell-further-democratizes-advanced-analytics, 2016.

[43] Documentation. "a new reality for oil & gas". https://www.cisco.com/c/dam/en_us/solutions/industries/energy/docs/OilGasDigitalTransformationWhitePaper.pdf, 2017.

[44] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[45] Eclipse IoT Working Group. IoT Developer Survey 2018. https://https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iot-developer-survey-2018, 2018.

[46] EsperTech. Esper. http://www.espertech.com/esper/, 2017.

[47] Facebook. Folly. https://github.com/facebook/folly#folly-facebook-open-source-library, 2017.

[48] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[49] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. Youprove: Authenticity and fidelity in mobile sensing. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[50] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul. Ariadne: Managing fine-grained provenance on data streams. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2013.

[51] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[52] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[53] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Securestreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2017.

[54] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[55] hortonworks. "iot and predictive big data analytics for oil and gas". https://hortonworks.com/solutions/oil-gas/, 2017.

[56] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing ARM trustzone. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security)*, 2017.

[57] iMatix Corporation. Zeromq. http://zeromq.org/, 2018.

[58] M. G. Institute. The internet of things: Mapping the value beyond the hype.

[59] Intel. Intel threading building blocks. https://software.intel.com/en-us/intel-tbb, 2017.

[60] Intel. "iot solutions for upstreamoil and gas". https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/oil-and-gas-iot-brief.pdf, 2017.

[61] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[62] Z. Jerzak and H. Ziekow. The debs 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2014.

[63] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2015.

[64] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endow.*, 2(2):1378–1389, 2009.

[65] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[66] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and

S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[67] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[68] A. Krettek and M. Winters. "the curious case of the broken benchmark: Revisiting apache flink® vs. databricks runtime". https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime, 2017.

[69] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*, 2016.

[70] Linaro. Op-tee: Open portable trusted execution environment. https://www.op-tee.org/, 2017.

[71] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for intel sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[72] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Conference on Security Symposium (USENIX Security)*, 2016.

[73] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[74] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[75] S. Madden. Intel lab data. http://db.csail.mit.edu/labdata/labdata.html, 2004.

[76] Magazine. "smart grids: Everything you need to know". https://www.cleverism.com/smart-grids-everything-need-know/, 2014.

[77] Magazine. "internet of things: A data-driven future for manufacturing". https://www.themanufacturer.com/wp-content/uploads/2017/01/IoT_FutureofManuf_ebook_final.pdf, 2017.

[78] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, 2005.

[79] 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 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[80] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys)*, 2008.

[81] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[82] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[83] Microsoft. Microsoft azure iot edge– extending cloud intelligence to edge devices. https://azure.microsoft.com/en-us/services/iot-edge/, 2017.

[84] Mohammad Marashi, Tech Crunch. Satellites are critical for IoT sector to reach its full potential. https://techcrunch.com/2017/06/08/satellites-are-critical-for-iot-sector-to-reach-its-full-potential/, 2017.

[85] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[86] S. Nastic, H. L. Truong, and S. Dustdar. A middleware infrastructure for utility-based provisioning of iot cloud systems. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[87] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[88] NXP Semiconductors. i.MX 7Dual Family of Applications Processors Datasheet, howpublished = https://www.nxp.com/docs/en/data-sheet/imx7dcec.pdf, year = 2017.

[89] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[90] Preferred networks. Sensorbee: Lightweight stream processing engine for iot. http://sensorbee.io/, 2017.

[91] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using SGX. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[92] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[93] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[94] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[95] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[96] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[97] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *Proceedings of the 11th Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[98] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *Proceedings of IEEE Pervasive Computing*, 14(2):24–31, 2015.

[99] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[100] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *Proceedings of IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[101] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[102] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *Proceedings of SIGMOD Rec.*, 34(3):31–36, 2005.

[103] A. Spark. "spark streaming programming guide". https://spark.apache.org/docs/latest/streaming-programming-guide.html, 2016.

[104] S. Sponseller. "the importance of the edge for the industrial internet of things in the energy industry". https://www.datascience.com/blog/predictive-analytics-in-industrial-iot, 2017.

[105] M. C. Stanley Zdonik, Michael Stonebraker. Streambase systems. http://www.tibco.com/products/tibco-streambase, 2017.

[106] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster. Styx: Stream processing with trustworthy cloud-based execution. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[107] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[108] Symantec. Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf, 2017.

[109] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Proceedings of Empirical Software Engineering*, 19(6):1665–1705, 2014.

[110] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[111] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu. WALNUT: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks. In *Proceedings of the 2nd Annual IEEE European Symposium on Security and Privacy*, 2017.

[112] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[113] Twitter. Heron. https://twitter.github.io/heron/, 2017.

[114] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[115] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[116] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[117] Wind River. SECURITY IN THE INTERNET OF THINGS – Lessons from the Past for the Connected Future. https://www.windriver.com/whitepapers/security-in-the-internet-of-things/wr_security-in-the-internet-of-things.pdf, 2017.

[118] X. Wu, R. Dunne, Q. Zhang, and W. Shi. Edge computing enabled smart firefighting: Opportunities and challenges. In *Proceedings of the 5th ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2017.

[119] 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 (S&P)*, 2015.

[120] D. Yarmoluk and C. Truempi. "predictive analytics in industrial iot". https://www.datascience.com/blog/predictive-analytics-in-industrial-iot, 2018.

[121] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[122] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[123] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. Case: Cache-assisted secure execution on arm processors. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[124] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves

Meni Orenbach
Technion

Yan Michalevsky
Anjuna Security

Christof Fetzer
TU Dresden

Mark Silberstein
Technion

Abstract

Hardware secure enclaves are increasingly used to run complex applications. Unfortunately, existing and emerging enclave architectures do not allow secure and efficient implementation of custom page fault handlers. This limitation impedes in-enclave use of secure memory-mapped files and prevents extensions of the application memory layer commonly used in untrusted systems, such as transparent memory compression or access to remote memory.

CoSMIX is a **Co**mpiler-based system for **S**ecure **M**emory **I**nstrumentation and e**X**ecution of applications in secure enclaves. A novel *memory store* abstraction allows implementation of *application-level secure page fault handlers* that are invoked by a lightweight enclave runtime. The CoSMIX compiler instruments the application memory accesses to use one or more memory stores, guided by a global instrumentation policy or code annotations without changing application code.

The CoSMIX prototype runs on Intel SGX and is compatible with popular SGX execution environments, including SCONE and Graphene. Our evaluation of several production applications shows how CoSMIX improves their security and performance by recompiling them with appropriate memory stores. For example, unmodified Redis and Memcached key-value stores achieve about 2× speedup by using a self-paging memory store while working on datasets up to 6× larger than the enclave's secure memory. Similarly, annotating a single line of code in a biometric verification server changes it to store its sensitive data in Oblivious RAM and makes it resilient against SGX side-channel attacks.

1 Introduction

Virtual Memory is integral to modern processor architectures. In addition to its primary role in physical memory management, it empowers developers to *extend* the standard memory layer with custom data storage mechanisms in software. For example, the memory-mapped file abstraction, which is broadly used, e.g., in databases [10, 5], relies on the OS's page fault handler to map a frame and populate it with the contents of a file. Replacing accesses to physical memory with file accesses requires no application code changes. Therefore, the ability to override page fault behavior has been essential for implementing a range of system services, such as memory compression [44], disaggregation [39, 75], distributed shared memory [36, 46] and heterogeneous memory support [37].

With the emergence of Software Guard Extensions (SGX) for Trusted Execution in Intel CPUs [16, 55], applications are increasingly ported to be entirely executed in *hardware-enforced enclaves* [58, 45, 23, 25]. The enclave hardware protects them from attacks by a powerful privileged adversary, such as a malicious OS or a hypervisor. A number of recent systems facilitate the porting to SGX by shielding unmodified applications in an enclave [21, 81, 18]. Unfortunately, these systems do not allow secure overriding of page fault handling in enclave applications. This drawback complicates porting a large class of applications that use memory-mapped files to SGX. Further, it prevents SGX applications from using security and performance enhancements, such as efficient memory paging [61] and Oblivious RAM (ORAM) side-channel protection [67, 11, 88] without intrusive application modifications. Our goal is to eliminate these constraints.

For example, consider the task of running an SQLite database that uses memory-mapped files in the enclave. The database file must be encrypted to ensure data confidentiality. Enabling in-enclave execution of SQLite therefore requires support for encrypted memory-mapped files, which in turn implies that the *page fault handler must be executed securely* as well. Unfortunately, hardware enclaves available today do not support secure page faults. Instead, existing solutions use workarounds, such as eagerly reading and decrypting the whole mapped file region into trusted enclave memory [18]. This solution does not scale to large files and lacks the performance benefits of on-demand data access.

We argue that the problem is rooted in the fundamental limitation of SGX architecture, which does not provide the mechanism to define *secure* page fault handlers. The upcoming SGX-V2 [54, 86, 43] will not solve this problem either. Moreover, we observe that existing and emerging secure enclave architectures [28, 4, 34] suffer from similar limitations(§2).

In this work, we build **CoSMIX**, a compiler and a lightweight enclave runtime that overcomes the SGX architectural limitations and enables secure and efficient extensions to the memory layer of *unmodified* applications running in enclaves. We introduce a *memory store, (*m*store)*, a programming abstraction for implementing custom memory management extensions for enclaves. The CoSMIX compiler automatically instruments application code to allocate the selected

variables and memory buffers in the *m*store, replacing the accesses to enclave memory with the accesses to the *m*store. The *m*store logic runs in the enclave as part of the application. The CoSMIX runtime securely invokes the *m*store memory management callbacks, which include custom page fault handlers. The page faults are semantically equivalent to hardware page faults yet are triggered by the CoSMIX runtime.

An *m*store can implement the missing functionalities that require secure page fault handlers. For example, it may provide the secure `mmap` functionality by implementing the page fault handler that accesses the file and decrypts it into the application buffer. A more advanced *m*store may add its own in-memory cache analogous to an OS page cache, to avoid costly accesses to the underlying storage layer. CoSMIX supports several types of *m*stores, adjusting the runtime to handle different *m*store behaviors while optimizing the performance.

CoSMIX allows the use of *multiple* m*stores* in the same program. This can be used, for example, to leverage both secure `mmap` *m*store and an ORAM *m*store for side-channel protection. Additionally, CoSMIX supports *stacking* of multiple *m*stores to enable their efficient composition and reuse. We design and prototype three sophisticated *m*stores in §3.2.5, and demonstrate the benefits of stacking in §4.5.

CoSMIX's design focuses on two primary goals: (1) minimizing the application modifications to use *m*stores and (2) reducing the instrumentation performance overheads. We introduce the following mechanisms to achieve them:

Automatic inference of pointer types. CoSMIX *does not* require annotating every access to a pointer. Instead, it uses inter-procedural pointer analysis [17] to determine the type of the *m*store (or plain memory) to use for each pointer. When the static analysis is inconclusive, CoSMIX uses *tagged pointers* [47, 74, 13] with the *m*store type encoded in the unused Most Significant Bits, enabling runtime detection (§3.3.1).

Locality-optimized translation caching. The *m*store callbacks interpose on memory accesses, which are part of the performance-critical path of the application. To reduce the associated overheads, we employ static compiler optimizations to reduce the number of runtime pointer type checks and *m*store accesses. These include loop transformations and a software *Translation Lookaside Buffer (TLB)* (§3.3.4). These mechanisms reduce the instrumentation overheads by up to two orders of magnitude (§4.2).

Our prototype targets existing SGX hardware and is compatible with several frameworks for running unmodified applications in enclaves [81, 1, 18]. However, CoSMIX makes no assumptions about enclave hardware. The CoSMIX compiler is implemented as an extension of the LLVM framework [48].

We prototype three *m*stores: Secure User Virtual Memory (SUVM) for efficient paging in memory-intensive applications [61], Oblivious RAM [78] for controlled side-channel protection, and a secure `mmap` *m*store that supports access to encrypted/integrity-protected memory-mapped files. We evaluate CoSMIX on the Phoenix benchmark suite [66], as well as on unmodified production servers: `memcached`, Redis, SQLite, and a biometric verification server [61]. The compiler is able to correctly instrument all of these applications, some with hundreds of thousands of lines of code (LOC), *without the need to manually change the application code*.

Our microbenchmarks using Phoenix with SUVM and secure `mmap` *m*stores show that CoSMIX instrumentation results in a low geometric mean overhead of 20%.

For the end-to-end evaluation, we run `memcached` and Redis key value stores on 600 MB datasets – each about 6× the size of the secure physical memory available to SGX enclaves. In this setting, SGX hardware paging significantly affects the performance. The SUVM [61] *m*store aims to optimize exactly this scenario. To use it, we only annotate the item allocator in `memcached` (a single line of code) and compile it with CoSMIX. Redis is compiled *without* adding annotations. The instrumented versions of both servers achieve about 2× speedup compared to their respective vanilla SGX baselines.

In another experiment, we evaluate a biometric verification server with a database storing 256 MB of sensitive data. We use the ORAM *m*store to protect it from SGX controlled side-channel attacks [87] that may leak sensitive database access statistics. We annotate the buffers containing this database (one line of code) to use ORAM. The resulting ORAM-enhanced application provides security guarantees similar to other ORAM systems for SGX, such as ZeroTrace [67], yet without modifying the application source code. ORAM systems are known to result in dramatic performance penalties of several orders of magnitude [26]. However, our hardened application is only 5.8× slower than the vanilla SGX thanks to the benefits of selective instrumentation enabled by CoSMIX.

To summarize, our contributions are as follows:

- Design of a compiler and an *m*store abstraction for transparent secure memory instrumentation (§3.2).
- Loop transformation and loop-optimized caching techniques to reduce the instrumentation overheads (§3.3.4).
- Seamless security and performance enhancement for unmodified real-world applications running in SGX, by enhancing them with the SUVM, ORAM and secure `mmap` *m*stores (§4).

2 Motivation

Enabling the use of custom page fault (PF) handlers in enclaves would not only facilitate porting of existing applications that rely on such functionality, but also enable a range of unique application scenarios, as we discuss next.

SUVM. The authors of Eleos [61] proposed Software User-space Virtual Memory (SUVM), which implements *exit-less memory paging in enclaves* and significantly improves the performance of memory-demanding secure applications. It keeps the page cache in the enclave's trusted memory, while the storage layer resides in untrusted memory whose contents are encrypted and integrity-protected.

ORAM. Oblivious RAM (ORAM) obfuscates memory access patterns by shuffling the physical data locations and re-encrypting the contents upon every access. As a result, an adversary observing the accessed locations learns nothing about the actual access pattern to the data [38]. Multiple ORAM schemes have been proposed over time [38, 84, 78, 88, 67, 33], and, ORAM was recently used to manually secure applications executing in SGX enclaves against certain side-channel attacks [88, 67, 33].

Both ORAM and SUVM are generic mechanisms that could be useful in many applications. Unfortunately, integrating them with the application logic requires intrusive code modifications. With the support for efficient and secure in-enclave PF handlers, we could add these mechanisms to existing unmodified programs, as we show in the current work.

Other applications include transparent compression for in-enclave memory, `mmap` support for encrypted and integrity-protected files, and inter-enclave shared memory, as well as various memory-management mechanisms [39, 75, 37].

Unfortunately, existing enclave hardware provides no adequate mechanisms to implement efficient and secure user-defined PF handlers, as we describe next.

2.1 Background: page-faults in enclaves

There are several leading enclave architectures: Intel SGX [16, 43], Komodo for ARM Trust Zone [34, 15], Sanctum [28], and Keystone [4]. Among these, only Intel SGX and Sanctum published support for paging. We briefly describe them below.
Intel SGX [16, 43, 55] supports on-demand paging between secure and untrusted memory. SGX relies on Virtual Memory hardware in X86 CPUs. When a PF occurs, the enclave *exits* to an *untrusted* privileged OS which invokes the SGX PF handler. The enclave execution resumes (via `ERESUME`) after the swapping is complete.

Since the PF handler is untrusted, the SGX paging is secured via SGX paging instructions. Specifically, `EWB` encrypts and signs the page when swapping the page out, whereas `ELDU` validates the integrity and decrypts when swapping it in. These instructions cannot be modified to perform other operations. They cannot change the internal SGX encryption key or modify the swapped page. In other words, they cannot act as a general-purpose secure PF handler.
Sanctum [28] supports per-enclave page tables and secure PF handlers. It uses a security monitor that runs at a higher privilege level than the OS and the hypervisor. Upon a PF, the enclave exits to the security monitor, which triggers the in-enclave secure PF handler.

2.2 Limitations of existing enclaves

Signal handling in SGX. Page fault handlers can be customized in userspace by registering a signal handler. SGX supports signal handlers in enclaves and works according to the following scheme (Figure 1): when an interrupt occurs, the enclave exits to the OS ❶. The OS takes control and performs an up-call to an untrusted user-space trampoline in the enclave's hosting process ❷. The trampoline re-enters the enclave by invoking the in-enclave signal handler ❸. After the signal handler terminates, the enclave exits ❹ and resumes execution of the last instruction ❺ via `ERESUME`.

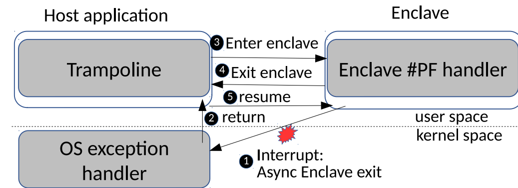


Figure 1: Execution of a signal handler in SGX

SGX: No secure page fault handlers. The SGX signal handling mechanism cannot guarantee secure execution of the handler itself. When the enclave is resumed after the PF, `ERESUME` replays the faulting memory access. Therefore, the enclave *cannot validate* that the signal handler was indeed executed correctly, or was executed at all. To the best of our knowledge, this problem will not be resolved in the next version of SGX [54, 86, 43].
SGX: Performance overheads. Even with hardware support for the secure signal handler, SGX has inherent architectural properties that will lead to significant performance penalties, rendering this mechanism unsuitable for customized application memory management. The architecture relies on the OS to manage the enclave's virtual memory. Furthermore, SGX may only run userspace code. Since the OS is untrusted, any secure page fault handling would inevitably follow the signal handling scheme depicted in Figure 1, namely, double enclave transition between the trusted and untrusted contexts.

We measure the latency of an empty SGX PF handler (access to a protected page) to be *11*μsec, which is more than 6$\times$ the latency of a signal handler outside SGX. For comparison, CoSMIX's software page fault handler is only 0.01 μ second, the cost of a single function call, which is three orders of magnitude faster than in SGX.

Further analysis shows that the signal latency is dominated by the latency of enclave transitions, which we measure to be 5.3μsec each [1] and stems from costly validation, register scrapping, TLB and L1 cache flushes [82, 43].
Other enclave architectures. Enclave transition overheads are pertinent to other enclave architectures. Komodo reports exit latency of 0.96μs [34]. Sanctum and Keystone do not disclose their enclave transition penalties, yet they describe similar operations performed when such transitions occur.

We conclude that *secure page fault handlers are not supported in SGX, and are likely to incur high-performance costs in other enclave architectures due to transition overheads.*

[1] This value is almost double the one reported in prior works [61, 85] because of the firmware update to mitigate the Foreshadow [82] bug.

2.3 Code instrumentation for enclaves

Instrumenting application memory accesses with the desired software PF handling logic is a viable option to achieve the functionality equivalent to the hardware-triggered PF handlers. Unfortunately, existing instrumentation techniques are not sufficient to achieve our goals, as we discuss below.

Binary instrumentation. Dynamic binary instrumentation tools [51, 57, 24, 52, 71], such as Intel PIN [51], enable instrumentation of memory accesses. Unfortunately, these tools have significant drawbacks in the context of in-enclave execution. For example, for PIN to work, all its libraries should be included in the enclave's Trusted Computing Base (TCB). Moreover, PIN requires JIT-execution mode to instrument memory accesses. Therefore, the enclave code pages should be configured as writable, which might lead to security vulnerabilities. Removing the write access permission from enclave pages will be supported in SGX V2, but doing so will require costly enclave transitions [43].

Static binary instrumentation tools do not suffer from these shortcomings. However, compared to the compiler-based techniques we propose in this work, they do not allow using comprehensive code analysis necessary for performance optimizations. Therefore, we decided against the binary instrumentation design.

Compiler-based instrumentation. The main advantage of this method is the ability to aggressively reduce the instrumentation overheads by using advanced source code analysis. On the other hand, the source code access requirement limits the applicability of this method. However, this drawback is less critical in the case of SGX enclaves because many SGX execution frameworks, such Panoply [77] and SCONE [18], require code recompilation anyway. Therefore, we opt for compiler-based instrumentation in CoSMIX.

3 CoSMIX Design

CoSMIX aims to facilitate the integration of different *m*stores efficiently into SGX applications. Our design goals are:

- **Performance.** Low overhead memory-access and software address translation.
- **Ease-of-use.** Annotation-based or automatic instrumentation without manual application code modification.
- **General memory extension interface.** Easy and modular development of application-specific memory instrumentation libraries.
- **Security.** Keep SGX security guarantees and small TCB.

Threat Model. CoSMIX is designed with the SGX threat model, where the TCB includes the processor package and the enclave's code. Additionally, we assume that the code running in an enclave does not contain memory vulnerabilities.

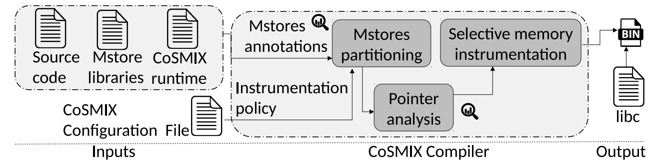


Figure 2: CoSMIX compilation overview. The compiler is guided by code annotations and global instrumentation policy.

3.1 Design overview

Compiler-based instrumentation. CoSMIX enables SGX developers to build custom *memory stores,* m*stores* that redefine the memory layer functionality of an instrumented program. To integrate one or more *m*stores into an application, the CoSMIX compiler automatically instruments the program (Figure 2). The developer may selectively annotate static variables or/and memory allocations to use different *m*stores, or define a global instrumentation policy. The compiler automatically instruments the necessary subset of memory accesses with the accesses to the corresponding *m*stores, and statically links *m*store libraries with the code.

The CoSMIX configuration file defines the instrumentation behavior. It specifies annotation symbols per *m*store, *m*store type (§3.2) and the instrumentation policy(§3.3).

3.2 *M*store abstraction

At a high level, an *m*store implements another layer of virtual memory on top of an abstract *storage layer*. An *m*store operates on pages, m*pages*, and keeps track of the *m*page-to-data mappings in its internal *m*page table. When an application accesses memory, the runtime invokes the *m*store's software *page fault handler*, retrieves the contents (e.g., for the secure `mmap` *m*store, it would read data from a file and decrypt), and makes it accessible to the application.

We distinguish between *cached* and *direct-access* *m*stores. A cached *m*store maintains its own *m*page cache to reduce accesses to the storage layer, whereas a direct-access *m*store does not cache the contents.

Figure 3 shows the execution of an access to a cached *m*store. The pointer access ❶ triggers the runtime call, which chooses the appropriate *m*store ❷ and checks the translation cache ❸. If the translation is not in the cache, the runtime invokes the *m*page fault handler ❹. The *m*store translates the pointer ❺, and either fetches the referenced *m*page from the page cache ❻, or retrieves it from the storage layer and updates the *m*page and translation caches ❼.

3.2.1 *M*store callbacks

Table 1 lists the callback functions *m*stores must implement.

Initialization/teardown. The *m*store is initialized at the beginning of the program execution and torn down when the pro-

Callback	Purpose
`mstore_init(params)/mstore_release()`	Initialize/tear down
`void* alloc(size_t s, void* priv_data)/free(void* ptr)`	Allocate/free buffer
`size_t alloc_size(void* ptr)`	Allocation size
`size_t get_mpage_size()`	Get the size of the *m*page
Direct-access *m*store	
`mpf_handler_d(void* ptr, void* dst, size_t s)`	*m*page fault on access to `ptr`, store the value in `dst`
`write_back(void* ptr, void* src, size_t size)`	Write back value in `src` to `ptr`
Cached *m*store	
`void* mpf_handler_c(void* ptr)`	*m*page fault on access to `ptr`, return pointer to *m*page
`flush(void* ptr, size_t size)`	Write the *m*pages in the range `ptr:ptr+size` to *m*store
`get_mstorage_base()/get_mpage_cache_base()`	Gets the base address of mstorage/*m*page cache
`notify_tlb_cached(void* ptr) / notify_tlb_dropped(void* ptr, bool dirty)`	The runtime cached/dropped the `ptr` translation in its TLB

Table 1: Compulsory *m*store callback functions

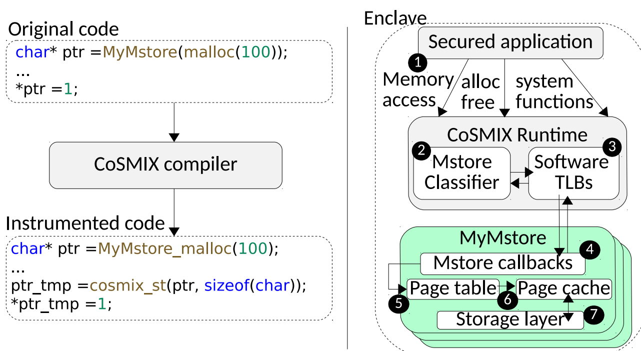


Figure 3: CoSMIX: code transformation and execution flow of access to a cached *m*store. See the text for explanation.

gram terminates. Importantly, the runtime flushes the *m*page cache when tear-down of cached *m*stores is called.

Memory allocation. The runtime delegates the memory allocation calls of the original program to the *m*store `alloc`.

3.2.2 Pointer access and *m*page faults

When the instrumented code accesses the *m*store, the runtime incurs an equivalent of a page fault, and invokes the respective callback in the *m*store, as discussed below.

Cached *m*stores. A cached *m*store *translates* the pointer to the *m*page inside its cache. `mpf_handler_c` returns the pointer into the *m*page that holds the requested data, allowing direct access from the code. For cross-page misaligned accesses, the runtime creates a temporary buffer and populates it by calling the `mpf_handler_c` for every page separately. For store access, the updates are written to both pages.

When the runtime determines that the code is accessing the same *m*page multiple times, it may *cache* the pointer to that *m*page in its private *TLB*. This avoids the address translation overheads for all but the first access to the page. To ensure that the *m*page is not swapped out by the *m*store logic, the runtime notifies the *m*store to pin the *m*page in the *m*page cache via `notify_tlb_cached`. The page is unpinned by `notify_tlb_dropped` when its translation is evicted from the TLB (see §3.3.4 for more details).

There can be multiple cached *m*stores in the same program, each with its own *m*page size. The runtime can query an *m*store page size using the `get_mpage_size` call, for example, to determine accesses to the same *m*page in the TLB.

A cached *m*store must implement the `flush()` callback to synchronize its *m*page cache with the storage layer. This callback is used, for example, to implement the `msync` call.

Direct-access *m*stores. Direct-access *m*stores have no cache and thus are easier to implement. The input pointer provided to the `mpf_handler_d` callback may be used without address translation, and at finer granularity not bound to the *m*page size. The runtime provides a thread-local intermediate buffer to store the accessed data. For loads, the program uses this buffer. For stores, the runtime writes the updated contents back to the *m*store using the `write_back` callback.

3.2.3 Thread safety and memory consistency

CoSMIX allows multiple threads to access the same *m*store, as long as the *m*store implementation is thread-safe.

For cached *m*stores, CoSMIX does not change the *m*store memory consistency model as long as the accesses are inside the same *m*page, and the *m*page size is 4KB or larger. The CoSMIX runtime does not introduce extra memory copies for such accesses and effectively inherits the *m*store memory consistency. In addition, the *m*store itself must ensure that the storage layer is accessed only via its *m*page cache, thereby preventing different threads from seeing inconsistent data.

This guarantee does not hold for direct-access *m*stores and misaligned cross-*m*pages. The primary implication is that hardware atomics will not work for such accesses.

We believe that this limitation does not affect most practical cases. The lack of cross-*m*page atomics support does not affect race-free programs synchronized via locks. This is because the intermediate buffer is written back to the *m*store immediately after the original store operation and thus will be protected by the original lock. We observed no cases of cross-*m*page atomics in the evaluated applications.

Today, the problem of misaligned cache accesses deserves special handling in compilers. For example, LLVM translates such accesses into the `XCHG` instruction [43]. CoSMIX must rely on a software solution, e.g., using readers-write locks per-*m*page. We defer this to future work.

3.2.4 Memory vs. file-backed *m*stores

The *m*store abstraction described so far instruments memory accesses alone. However, it is insufficient to enable implementation of memory-mapped files. For example, consider a program that uses `mmap` to access a file, and then calls `fsync`. It will not work correctly because `fsync` is not aware of the *m*store's internal cache. Specifically, all the I/O operations on files managed by a file-backed *m*store must interact with its *m*page cache to avoid data inconsistency.

We define a *file-backed* cached *m*store type, which implements all the callbacks of memory-backed *m*stores, but additionally overrides the POSIX file API that interacts with the page cache, e.g., `open, close, read, write, msync` (§3.3). Direct-access *m*stores do not have internal caches; thus they can be used with files without overriding file I/O operations other than `mmap` itself.

3.2.5 *M*store examples

CoSMIX provides a set of reusable building blocks for *m*stores, such as a slab memory allocator, spinlocks, a generic *m*page table, and an *m*page cache, all with multi-threading support, which we use to implement the *m*stores below.

SUVM *m*store. SUVM allows exit-less memory paging for enclaves by keeping a page table, page cache and a fault handler as part of the enclave. We implement SUVM from scratch as a cached memory-backed *m*store, using CoSMIX's generic *m*page table and *m*page cache. The `alloc` function returns a pointer to the storage layer in untrusted memory. Upon `mpf_handler_c`, the *m*store checks whether the needed *m*page is already cached in the *m*page table. If not, it reads the *m*page's contents from the storage layer, decrypts and verifies its integrity using a signature maintained for every *m*page, and finally copies it to the *m*page in the page cache. When the *m*page cache is full, the *m*store evicts pages back into the storage layer.

Secure `mmap` *m*store. This *m*store enables the use of memory-mapped encrypted and integrity-protected files in enclaves. We support private file mapping only and leave cross-enclave sharing support for future work. This is a cached file-backed *m*store that maintains its own *m*page table and *m*page cache.

The `alloc` callback is invoked by the runtime upon the `mmap` call in original code. `alloc` records the mapping start offset in the file and the access flags in an internal table. It then returns a pointer with the requested offset from a unique file base address. This address is internally allocated by the *m*store and used as isolated address space for each file.

The `mpf_handler_c` callback translates the given pointer to the respective file. If the contents are not available in the *m*page cache, the data is fetched from the file using a `read` system call, followed by decryption and integrity check.

Oblivious RAM *m*store. ORAM obfuscates memory access patterns by shuffling and re-encrypting the data upon every data access. The ORAM *m*store streamlines the use of ORAM in enclaves. It allows the developer to allocate and access the buffers that store sensitive data in an ORAM, thereby protecting the program against controlled side-channel attacks [87].

We implement a direct-access memory-backed ORAM *m*store. This is because if it were cached, the accesses to the cache would be visible to the attacker, compromising the ORAM security guarantees. Our ORAM *m*store addresses a threat-model similar to ZeroTrace [67], yet without leakage protection in the case of enclave shutdowns. Specifically, all the instrumented memory accesses destined to the ORAM *m*store become oblivious, such that an adversary cannot learn the actual memory access pattern. We implement the Path ORAM algorithm [78] and ensure oblivious accesses to its position map and stash using the *cmovz* instruction.

We store the Path-ORAM tree in a contiguous buffer within the enclave trusted memory. This eliminates the need to implement block encryption and integrity checks as part of the ORAM *m*store since SGX hardware does exactly that.

The `alloc` function allocates the requested number of blocks in ORAM and registers them with the ORAM module. It returns an address from a linear range with a hard-coded base address, which is used only to compute the block index.

The `mpf_handler_d` callback translates the address to the requested block index and invokes the ORAM algorithm to obliviously fetch the requested memory block to a temporary buffer. Loads are issued from this buffer and stores are appended with the `write_back` callback.

3.2.6 Stacking *m*stores

The *m*store abstraction makes it possible to *stack* different *m*stores. Stacking allows one *m*store to invoke another *m*store recursively. We denote by A→B an *m*store A that internally accesses its memory via *m*store B.

Consider, for example, ORAM→SUVM. The motivation to stack is when the ORAM *m*store covers a region that is larger than the enclave's physical memory. Since SUVM optimizes the SGX paging mechanism, stacking ORAM on top of SUVM improves ORAM's performance (§4.5).

To create a new A→B *m*store, the developer simply annotates A's storage layer allocations with B's annotations. CoSMIX instruments all these accesses appropriately.

Stacking the ORAM *m*store on top of any *m*store that maintains data confidentiality does not compromise the ORAM access pattern obfuscation guarantees, as ORAM protocols consider the backing store to be untrusted [78]. Therefore ORAM→SUVM would maintain data-oblivious access.

However, the stacking → operator is *not commutative* from the security perspective: SUVM→ORAM would result in the SUVM *m*store caching *m*pages fetched obliviously by the ORAM *m*store, thereby leaking the access patterns to these *m*pages and compromising ORAM's security guarantees.

3.3 CoSMIX compiler and runtime

The instrumentation compiler modifies the application to use *m*stores and is guided by code annotations or/and a global instrumentation policy. The compiler needs to instrument four different types of code: (1) memory accesses; (2) memory management functions; (3) file I/O operations for file-backed *m*stores; (4) `libc` library calls.

Instrumentation policy. A developer may annotate any static variable declaration or memory allocation function call. Annotations allow instrumentation of a subset of the used buffers to reduce instrumentation overheads. Alternatively, a global instrumentation policy specifies compile-time rules applied to the whole code base (e.g., instrument all calls to `malloc`), or run-time checks injected by the compiler (e.g., using *m*store for large buffers above a certain threshold). A global policy serves for bulk operations on large code bases, such as adding SUVM *m*store to Redis sources with over 130K LOC (§4.4).

Similarly, for file-backed *m*stores, a global policy may limit the use of the *m*store to specific files or directories.

3.3.1 Pointer access instrumentation

Static analysis. The compiler uses static build. Therefore, it can conservatively determine the subset of operations that must be replaced with *m*store-related functions at compile time and eliminate the instrumentation overhead for such cases. Trivially, the compiler may replace an annotated call to `malloc` with the `alloc` callback of the requested *m*store. A much more challenging task, however, is to determine *the type of the* m*store (if any) to use for every pointer in the program*.

For this purpose, we use Andersen's analysis [17] to generate inter-procedural point-to information. In a nutshell, CoSMIX first parses all instructions and generates a graph with pointers as nodes and their constraints (e.g., assignment or copy) as edges. The graph is then processed by a constraint solver which outputs the set of points-to-information.

When instrumenting memory accesses, CoSMIX can use this information to determine whether the pointer may alias to a specific *m*store pointer.

Runtime checks and tagged pointers. CoSMIX's pointer analysis is sound but incomplete; therefore it requires runtime decisions for ambivalent cases. We use tagged pointers [47, 13, 74, 31] to determine pointer type at runtime. Each *m*store is assigned a unique identifier, stored in unused most significant bits of the pointer virtual address. For instrumented allocations, the runtime adds this identifier to the returned address from the *m*store allocation. For external function calls and memory accesses, the runtime checks the tag, strips it from the pointer, and invokes the callback of the respective *m*store if necessary.

Tagged pointers vs. range checks. One known limitation of tagged pointers is that the application code might reset the higher bits of a pointer. Prior work [47] and our own experience suggest that this is rarely the case in practice. An alternative approach is to differentiate between *m*stores by assigning a unique memory range to each. Using tagged pointers turned out to be faster in our experience because the range check requires additional memory accesses.

3.3.2 Memory management and file I/O calls

The compiler replaces all the memory management operations selected by the instrumentation policy with the calls to the runtime that invokes the appropriate *m*store callbacks.

Similarly, file I/O operations are replaced with runtime wrappers. On `open`, the runtime determines whether to use an *m*store with the current file and registers its file descriptor. An I/O call using this file descriptor will be redirected to the respective *m*store.

3.3.3 `libc` support

Invoking an uninstrumented function on an *m*store pointer would result in undefined behavior. We assume that most application libraries are available at compile time. However, `libc` is not instrumented and we provide wrappers instead, similarly to other works [47, 59].

There are two main reasons to not instrument `libc`. First, doing so would create a bootstrapping problem since *m*stores might use `libc` calls. Second, SGX runtimes such as SCONE [18] use proprietary, closed-source versions of `libc`. Wrapping `libc` functions allows CoSMIX to be portable across multiple enclave execution frameworks.

CoSMIX provides wrappers for most popular `libc` functions (about 80), which suffices to run the large production applications we use in our evaluation. Adding wrappers for more functions is an ongoing effort.

In addition to stripping the pointer tag, these wrappers must guarantee access to virtually contiguous input/output buffers from uninstrumented code, instead of using *m*store *m*pages. Thus, where necessary, the wrappers use a temporary buffer in regular memory to stage the *m*store buffer before the library call and write it back to the *m*store after the call.

3.3.4 Translation caching

Minimizing the instrumentation overhead is a fundamental requirement for CoSMIX. The overheads are caused mainly by runtime checks of the pointer type and invocation of the *m*store logic on memory accesses.

To reduce these overheads, CoSMIX first runs aggressive generic code optimizations, reducing memory accesses. It

also avoids invoking *m*store page fault handlers for recurrent accesses to the same *m*page.

Opportunistic caching. We introduce a small (one cache line) TLB stored in the thread-local memory. This TLB is checked upon each access to the *m*store. The runtime pins the page in the *m*store while the *m*page translation is cached, and unpins when it is flushed. To support multiple threads, the TLB notification callbacks use a reference counter for each *m*page. The *m*page can be evicted if the counter drops to zero, eliminating the need for explicit TLB invalidation. We choose small TLB size (5) for its low lookup times. Increasing the size did not improve performance in our workloads.

Translation caching in loops. The TLB captures the locality of accesses quite effectively, but in loops, the performance can be further improved by transforming the code itself to use the *m*page base address without checking the TLB.

For example, in the case of an array allocated in an *m*store and sequentially accessed in a tight loop, most accesses to the *m*store are performed within the same *m*page. Therefore, replacing the code in the loop to check the TLB only at m*page boundaries* would result in near-native access latency.

To perform this optimization, CoSMIX has to (a) determine the iterations in which the same *m*page gets accessed, and (b) determine the pointer transformation across the iterations. For (b), CoSMIX uses the scalar evolution analysis [83] in the compiler to find predictable affine transformations for the pointers used in the loop. For (a) it injects a code that determines the number of iterations where the cached translation hits the same *m*page, recomputing the new base pointer and dropping the translation from the TLB when crossing into a new *m*page. Finally, it replaces the original accesses to the *m*store with the accesses to the *m*page's base pointer with the offset, which is updated across the loop iterations according to the determined transformation.

3.4 Discussion

Security guarantees. CoSMIX itself does not change the security of SGX enclaves. Its runtime neither accesses untrusted memory nor leaks secret information from the enclave. However, the security of *m*stores depends on their implementation: SUVM has the same security as SGX paging [61] and the ORAM *m*store introduces a controlled side-channel [87] protection mechanism not available in the bare-metal SGX [67]. We note, however, that CoSMIX does not guarantee that the code using an *m*store will indeed maintain that *m*store's security properties. For example, in case of the ORAM *m*store, user code must not use sensitive values read from ORAM in a data/control dependent manner because doing so might break the data pattern obfuscation [65].

Other *m*store applications. *M*stores are general and can be used to implement many other useful extensions. For example, implementing bounds checking for 32-bit address space enclaves as in SGXBounds [47] becomes easy. All it takes is adding an extra 4 bytes for each buffer allocation to store the lower bounds of the object and tag the pointer's highest 32 bits with its lower bounds' address. Then, every memory access is instrumented to check these bounds. The SGXBounds *m*store can implement this logic in its callback functions.

Another useful application is transparent inter-enclave shared memory, which may enable execution of multi-socket enclaves and support for secure file sharing.

Eleos vs. CoSMIX. The starting point of our design was Eleos [61]. There, the authors introduce *s*pointers, which are similar to C++ smart pointers. In Eleos all necessary memory accesses are replaced with *s*pointers and translations are cached in the *s*pointers themselves. On every access, *s*pointers perform bound checking, to make sure that pointer arithmetic on the *s*pointer did not cross to a new page. However, the *m*page bound check impacts performance greatly, even when the caching is limited to the scope of a function. Second, maintaining static translation for *m*store pointers complicated the design. For example, pointer-to-integer casts had to be invalidated, forcing a reverse mapping in *m*stores.

A key lesson from CoSMIX is that *caching the translations only in cases of high access locality is enough to leverage the performance benefits and simplify the design.*

Limitations. Inline assembly snippets, while quite rare, cannot be easily supported. CoSMIX considers them as an opaque function call. It injects code to check whether passed arguments are *m*store pointers. If so, CoSMIX aborts the program and notifies that manual instrumentation is necessary.

Hardware extensions. We hope that CoSMIX will motivate hardware developers to support *secure in-enclave fault handling*. This functionality would allow enclaves to control the execution flow of the page faults. For example, an enclave might refuse to resume execution after a fault unless a correct secure fault handler has been invoked. As a result, secure page faults would allow extending enclaves with cached *m*store functionality, such as the secured `mmap` provided by CoSMIX, albeit at a significantly higher performance costs due to transitions to/from untrusted mode. Moreover, hardware support for secure fault handlers would enable paging of code pages not supported by CoSMIX.

However, direct *m*stores such as ORAM cannot be supported in the same way, since they invoke the fault handler for every memory access. Therefore, good performance could be achieved only by much more intrusive modifications that would avoid enclave mode transitions. Additionally, enclave hardware evolves slowly. For example, the SGX2 specification was published in 2014, yet is still not publicly available in mainstream processors [20]. CoSMIX on the other hand, can be used to enhance enclaves' functionality today.

4 Evaluation

Implementation. CoSMIX implementation closely follows its design [2]. The compiler prototype is based on the LLVM 6.0 framework and is implemented as a compile-time module pass, consisting of 1,080 LOC. We compile applications using the Link-Time Optimization (LTO) feature of LLVM and pass them as inputs to the compiler pass.

CoSMIX uses the SVF framework [79, 80] to perform Andersen's pointer analysis with type differentiation. CoSMIX runtime is written in C++ and consists of 1,600 LOC. CoSMIX's configuration file is `JSON` formatted and CoSMIX uses JsonCpp [3] to parse it. All *m*stores are written in C++. Their implementation follows the design described in Section 3. The implementations of SUVM, ORAM and `mmap` *m*stores are 935 LOC, 551 LOC, and 1,108 LOC respectively. `mmap` *m*store leverages the SCONE file shields, which override the `read/write` calls in `libc` to implement integrity checks and encryption for file I/O operations. This is one of the examples when our design choice to use `libc` wrappers rather than `libc` instrumentation pays off.

Setup. Our setup comprises two machines: server and client. The client generates the load for the server under evaluation. The client and the server are connected back-to-back via a 56Gb Connect-IB operating in IP over Infiniband [27] (32Gbps effective throughput) to stress the application logic and avoid network bottlenecks.

For the server, we use Dell OptiPlex 7040, with Intel Skylake i7-6700 4-core CPU with 8 MB LLC, 16 GB RAM, and 256 GB SSD drive, Ubuntu Linux 16.04 64-bit, Linux 4.10.0-42, and Intel SGX driver 2.0 [2]. We use LLVM 6.0 to compile the source code. As recommended by Intel, we apply L1TF microcode patches [9]. The client runs on a 6-core Intel Xeon CPU E5-2620 v3 at 2.40GHz with 32GB of RAM, Ubuntu 16.04 64-bit, Linux 4.4.0-139.

Methodology. Unless otherwise specified, we run all the workloads in SGX using SCONE [18]. We run each test 10 times and report the mean value, with the standard deviation below 5%. We compile all workloads as follows: (1) compile to LLVM IR with full optimizations (-O3) and invoke LLVM's IR linker to link all IR files; (2) invoke CoSMIX LLVM pass for code instrumentation; (3) use the SCONE compiler to generate an executable binary linked with its custom libc. We skip step (2) when compiling the baseline.

Summary of workloads. We evaluated several production applications and benchmarks, detailed in Table 2. CoSMIX successfully instruments large code bases using only *a few or no code annotations, and no source code changes*.

[2]CoSMIX source code is publicly available at `https://github.com/acsl-technion/cosmix`.

Workload	LOC	Changed LOC	*m*store
`memcached` [35]	15,927	1	SUVM
Redis [8]	123,907	0	SUVM
SQLite [62]	134,835	2	secure `mmap`
Phoenix suite [66]	1,064	1/bench	SUVM ‖ secure `mmap`
Face verification [61]	700	1	SUVM → ORAM

Table 2: Summary of the evaluated workloads. ‖ - side-by-side, → - stacked. LOC includes all statically linked libraries.

Data size	Baseline	SUVM	ORAM
16 MB	0.7μsec	0.9μsec ($-1.28\times$)	32.3μsec ($-46.1\times$)
256 MB	14.4μsec	1.5μsec ($9.6\times$)	590.6μsec ($-41\times$)
1GB	19.9μsec	1.6μsec ($12.4\times$)	1.23*m*sec ($-61.8\times$)

Table 3: *m*store latency to fetch a 4 KB page. Baseline is native memory access.

4.1 *M*store performance

First, we measure the latency of random accesses to *m*stores and compare them to native memory accesses in the enclave. We evaluate scenarios with small and large memory usage where the latter causes SGX page faults. We report the results for SUVM and ORAM *m*stores and exclude the `mmap` *m*store because it is similar to SUVM. We measure the average latency to access random 4 KB pages over 100k requests.

Table 3 shows the results. For small datasets, SUVM incurs low overhead compared to regular memory accesses. However, for large data sets which involve SGX paging, it is about $10\times$ faster. This is because SUVM optimizes the enclave paging performance by eliminating enclave transitions [61]. As expected, ORAM *m*store access is between $30\times$ to $60\times$ slower than the baseline, even when covering a small range of 16 MB. This result indicates that selective instrumentation for ORAM is essential to achieve practically viable systems.

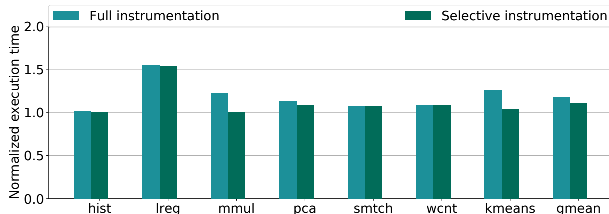
4.2 Instrumentation and *m*store overheads

We instrument all seven benchmarks in the Phoenix suite [66] to measure CoSMIX's instrumentation overheads. Each benchmark is small, making the results easier to analyze.

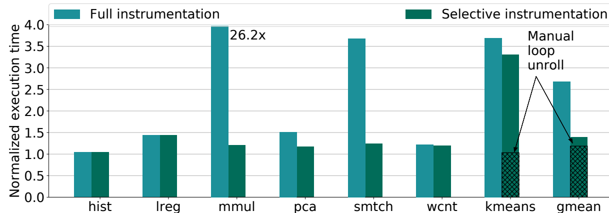
We evaluate 4 configurations: (1) Full automatic instrumentation using SUVM and `mmap` *m*store. Both *m*stores are run side-by-side because Phoenix uses both dynamic allocations and memory-mapped files. (2) Same but with an *empty m*store. All the pointers are instrumented, but the *m*store logic is not invoked. (3) Selective instrumentation where we manually annotate only the inputs. (4) Same but with an empty *m*store. For benchmarks that use `mmap`, the baseline reads the entire input file to memory.

Measuring an empty *m*store allows us to distinguish between overheads of pointer instrumentation and *m*stores.

To focus solely on the CoSMIX overheads, we use the small dataset shipped with Phoenix so no paging operation will occur. The only exception is the histogram benchmark, for which we synthetically resize the dataset to 25 MB. We exclude *m*store initialization and preload all the datasets into



(a) Instrumentation-only cost (empty mstores).



(b) Full cost with mstores.

Figure 4: CoSMIX instrumentation overheads for the Phoenix suite running in the enclave normalized to the execution of non-instrumented binaries. Lower is better.

SGX memory, both in the baseline and in CoSMIX measurements, to stress the runtime components of the system.

Often *m*stores can be tuned to reduce the translation overhead, by increasing the *m*store page size. As a result, the accesses in a loop might touch fewer pages, enabling more efficient use of CoSMIX's TLB. Therefore, we manually tune the page size, setting it to 256 KB for kmeans, 16 MB for word count, 64 MB for lreg, and 4 KB for the rest of the benchmarks, as in all the other experiments.

Figure 4 shows the results. Figure 4b shows the overhead for both CoSMIX's instrumentation and *m*store logic. Figure 4a excludes the *m*store logic. In each figure, the rightmost bar refers to the selective instrumentation of accesses to the input data alone, and the other bar refers to full instrumentation of all dynamic allocations and `mmap` calls.

Instrumentation overheads. The runtime overheads excluding *m*stores are relatively small, with an average (geomean) of 17% for full instrumentation and 10% for the input instrumentation alone, with the worst case of 50% in lreg.

Full instrumentation. With the full instrumentation, *m*store logic dominates the runtime overheads, ranging from almost none for histogram to 26× for matrix multiplication. Such variability stems from the different ways memory is accessed. Specifically, if the program exhibits poor access locality, or the CoSMIX compiler fails to capture the existing locality in a loop, the runtime will not be able to optimize out the calls to the *m*store inside the loop, resulting in high overheads.

Selective instrumentation. Instrumenting only the input buffers results in dramatically lower overheads, ranging from 5% to 15%. The only pathological case is kmeans, where the CoSMIX compiler fails to optimize accesses to the multidimensional input array because the inner array is too short. Unrolling this loop reduces the overhead to about 5%. We plan to add automatic optimizations for this case in the future.

	CoSMIX secure `mmap`	`read` **no cache**	`read` **1 MB cache**	`read` **16 MB-60 MB cache**
Query latency	2.4μsec	10.7μsec	4.5μsec	1.7μsec
Speedup		4.4×	1.8×	−1.4×

Table 4: SQLite performance with secure `mmap` *m*store.

Contribution of CoSMIX optimizations. We measure the performance of the *selective* instrumentation while disabling the runtime TLB and compiler loop optimizations (§3.3.4). We find that these two features *are essential* to make CoSMIX practical and keep the instrumentation overheads low. The slowdown ranges from 4× for word count and kmeans to 55× for histogram and 197× for lreg, making the system unusable. In comparison, the optimized version brings the overheads down to 4% and 44% for histogram and lreg. The geomean of the unoptimized selective instrumentation is 16.4× compared to 20% with the optimizations enabled.

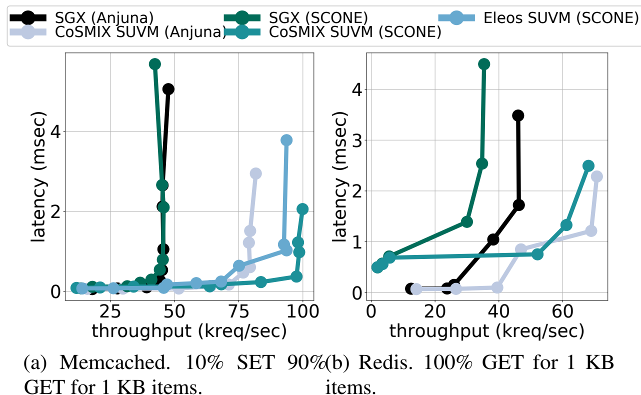
CoSMIX overheads for non-enclave execution. For completeness we perform the same experiment with Phoenix but now outside the enclave. As expected, the relative overheads increase as compared to the in-enclave execution, with up to 50% geometric mean slowdown when using selective instrumentation with SUVM and `mmap` *m*stores and up to 25% when using an empty *m*store. We attribute this discrepancy to SGX's memory encryption engine [40], which provides confidentiality and integrity to memory accesses and therefore offsets the CoSMIX instrumentation overheads.

4.3 Secure `mmap` with SQLite

SQLite is a popular database, but running it in SGX with `mmap` while providing encryption and integrity guarantees for the accessed files is not possible today. To run SQLite with `mmap` support, we use the secure `mmap` *m*store. We configure the *m*page size to be 4 MB. We use SQLite v3.25.3, and evaluate it with kvtest [10], shipped with SQLite to test read access latency to DB BLOBs. We use a database stored in a 60 MB file holding 1 KB BLOBs. The database is sized to fit in SGX physical memory. This allows us to focus on the evaluation of the file access logic rather than SGX paging (refer to §4.4 for paging evaluation).

As a baseline, we evaluate SQLite with its internal backend that uses `read/write` calls instead of memory-mapped files. In this configuration, SQLite implements its own optimized page cache for data it reads from files. In the evaluation, we vary the SQLite page cache size from disabled (1 KB) to 60 MB (no misses). We measure the average latency of 1 KB random read requests over 1 million requests.

Results. CoSMIX enables execution of an unmodified SQLite server that uses `mmap` to access encrypted and integrity-protected files. Such execution was not possible without CoSMIX. Moreover, as we see in Table 4, the secure *m*store enables 4.4× faster queries compared to the SQLite without



(a) Memcached. 10% SET 90% GET for 1 KB items.

(b) Redis. 100% GET for 1 KB items.

Figure 5: Performance improvement using the SUVM *m*store in production key-value stores, each using a 600 MB database (6× the size of SGX secure memory)

page cache. This case illustrates the CoSMIX performance benefits for applications that do not implement their own optimized page cache.

On the other hand, enabling the SQLite page cache allows us to evaluate the instrumentation overheads. This is, in fact, a conservative estimate, because the baseline is hand-optimized and implements the necessary functionality by itself, versus the general instrumentation and generic CoSMIX's page cache. Even in this worst-case scenario, CoSMIX is only about 40% slower than SQLite.

4.4 Optimizing memory-intensive workloads with the SUVM *m*store

To demonstrate CoSMIX's support for different SGX execution frameworks, we run the following experiments both in SCONE [18] and Anjuna [1].

We use CoSMIX to accelerate SGX execution of applications with a large memory footprint. We choose Redis and `memcached` key-value stores as representatives. Both run with data sets of 600 MB – about 6× larger than the SGX enclave-accessible physical memory. These applications experience a significant slowdown due to SGX paging overheads. The goal is to reduce these overheads using the SUVM *m*store.

Memached. `memcached` uses a slab allocator to manage its memory. We annotate a *single line of code* where the memory is allocated, making SUVM *m*store manage all items.

We evaluate `memcached` v1.4.25 [35] using the memaslap load generator shipped with libmemcached v1.0.18 [6]. Our workload consists of 10% SET and 90% GET requests for 1 KB items (key+value) as used in prior works [61], with requests uniformly distributed to stress the memory subsystem.

We compare the instrumented `memcached` with SUVM to native SGX execution. In addition, we run a *manually optimized* version of `memcached` with the SUVM used in Eleos [61]. Notably, in Eleos, the authors changed `memcached` internals to create a shadow memory buffer for the slab allocator. This is an intrusive change that CoSMIX eliminates completely. All runs are performed on 4 cores. Figure 5a shows that SUVM *m*store boosts the throughput by 1.9× and 2.2× compared to native SGX execution in Anjuna and SCONE respectively. The difference between the frameworks correlates with the relative time each of them spends resolving page faults. Interestingly, the CoSMIX version is about 7% *faster* than Eleos thanks to its compiler optimizations.

Redis. Manually annotating Redis with its 130 KLOC would be too tedious. Further, its memory management involves too many allocations, making annotation challenging. Therefore, we use automatic instrumentation *without code changes*.

To achieve high performance, we leverage CoSMIX's ability to perform conditional instrumentation. Specifically, CoSMIX introduces *runtime checks* that determine whether to allocate a buffer in an *m*store or in regular memory based on the requested allocation size. In Redis, we configure the policy to redirect all allocations in the range of 1 KB-10 KB to the SUVM *m*store. The intuition is to use SUVM only for keys and values and keep native memory accesses for the rest.

We use Redis v5.0.2 [8], and evaluate it using the memtier v1.2.15 official RedisLab load generator [7]. We configure memtier to generate uniformly distributed GET requests for 1 KB items as used in prior works [18].

Figure 5b shows that Redis with CoSMIX achieves about 1.6× and 2× higher throughput compared to native SGX execution in Anjuna and SCONE respectively. *These results demonstrate the power of CoSMIX to improve the performance of an unmodified production system.*

4.5 Protecting data with the ORAM *m*store

Selective instrumentation capabilities in CoSMIX are particularly useful when using heavyweight *m*stores such as ORAM. ORAM is known to dramatically affect performance (Table 3).

We use ORAM to protect a face verification server [61] against controlled side-channel attacks [87] on its data store.

The server mimics the behavior of a biometric border control server. It stores a database with sensitive face images. When a person passes through border control, the client at the border kiosk queries the server whether the image of the person in the database matches the one taken at the kiosk.

The implementation stores the images in an array. The server fetches the image from the array and compares it with the input image, using the LBP algorithm [12]. This implementation is vulnerable to controlled channel attacks which leak the access pattern to SGX memory pages. Thus, an attacker may observe page access frequency and learn when a person passes through border control. Note that existing defenses against controlled channel attacks would be ineffective since they cannot handle legitimate demand paging [76, 60].

We use a database with 1,024 256 KB images from the Color FERET dataset [63], totaling 256 MB of sensitive data. We annotate the allocation of the database array to use ORAM

	Native SGX	ORAM	ORAM→SUVM
Throughput(req/sec)	203.1	23.4	34.7
Slowdown		8.6×	5.8×

Table 5: Selective instrumentation of face verification server.

(1 LOC) and compile with CoSMIX. As a result, the application accesses the sensitive data obliviously. In the experiments we configure the server to use 1 thread and the load generator to issue random requests, saturating the server.

We report the throughput achieved in Table 5. It shows that the ORAM *m*store introduces an 8.6× slowdown. We note that for a dataset of 256 MB, the ORAM *m*store overhead is about 41× more than native access, as reported in Table 3. This shows that selective instrumentation may make ORAM an attractive solution for some systems. However, with CoSMIX we can further reduce paging penalties.

ORAM→SUVM stacking. Although the application data is only 256 MB, PathORAM consumes about 860 MB due to its internal storage overheads. As a result, the SGX paging significantly affects the performance.

To optimize, we create a new ORAM→SUVM *m*store by stacking ORAM on top of SUVM. Only 1 LOC in the ORAM *m*store is annotated. The use of the combined ORAM→SUVM *m*store improves the overall performance by 1.5× compared to the ORAM *m*store alone (Table 5). Overall, selective instrumentation and *m*store stacking result in a relatively low, 5.8× slowdown of the oblivious system compared to native SGX execution. This performance might be acceptable for practical ORAM applications and requires no code changes.

5 Related Work

Enclave system support. Recent works proposed systems to protect unmodified applications using enclaves [18, 21, 77, 81]. Other works proposed enclave enhancements [72, 47, 49], such as memory safety, ASLR, and enclave partitioning. Complementary to these works, CoSMIX provides system support for modular extensions to unmodified enclave applications.

Trusted execution environments. Previous works proposed different systems to protect applications from a malicious OS [30, 29, 42, 16, 15]. InkTag [42], for example, offers secured `mmap` service to applications; however, it relies on a trusted hypervisor with para-verification. CoSMIX puts its root of trust in hardware enclaves to implement in-enclave secure fault handlers.

Controlled side-channel mitigation. Previous works suggested the use of ORAM in SGX to improve its security [65, 67, 33, 88, 11, 50]. We believe that CoSMIX will allow the use of ORAM in many more applications via lightweight annotations. More efficient systems for mitigating the controlled side-channel attack have been proposed [22, 60, 76, 30]. For example, Apparition [30] uses Virtual Ghost [29], a compiler-based virtual machine, to restrict OS access to the page table. However, these systems do not support demand paging. CoSMIX protects applications that rely on demand paging from both a malicious OS and physical bus snooping attacks using the ORAM *m*store.

Customizing applications via paging mechanisms. Previous systems proposed using page faults to improve performance and enable quality of service functionality in OS, GPUs, and enclaves [41, 61, 74, 14]. CoSMIX enables using similar enhancements in secure enclave systems.

Recent works take advantage of the RDMA infrastructure to enable efficient and transparent access to remote memory by customizing the OS page fault handler [39, 75, 53, 37]. For example, LegoOS [75] uses paging to simulate an *ExCache* for disaggregated memory support. CoSMIX takes a similar approach for extending secure enclaves.

Software-based distributed shared memory systems were proposed to customize memory access logic across remote machines [68, 69, 70, 64, 19, 56, 32]. These systems either use the page fault handler, runtime libraries or instrumentation of applications. CoSMIX is inspired by these systems; however, the *m*store abstraction is more general and can support different memory access semantics, mixed or stacked in the same application.

Memory instrumentation. Much work has been done on instrumenting memory accesses using binary instrumentation tools [51, 24, 57, 52, 71] and compiler-based instrumentation [73, 47, 13, 31]. CoSMIX enables low-overhead selective memory instrumentation, specifically tailored for enclaves.

6 Conclusions

CoSMIX is a new system for modular extension of the memory layer in unmodified applications running in enclaves. It sidesteps the lack of hardware support for efficient and secure page fault handlers in existing architectures. CoSMIX enables low-overhead and selective instrumentation of memory accesses via a simple yet powerful *m*store abstraction.

We show how compilation with CoSMIX both speeds up execution and adds protection to production applications. We believe that *m*stores may become a useful tool for facilitating the development of new secured systems.

7 Acknowledgments

We would like to thank our shepherd John Criswell for his valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18), the Israeli Innovation Authority Hiper Consortium, the Technion Hiroshi Fujiwara Cybersecurity center, and Intel ICRI-CI Institute grant, and the feedback from the Intel SGX SEM group. Meni Orenbach was partially supported by HPI-Technion Research School.

References

[1] Anjuna. https://www.anjuna.io. Accessed: 2019-01-01.

[2] Intel SGX Linux Driver. https://github.com/intel/linux-sgx-driver. Accessed: 2018-12-06.

[3] JsonCpp. https://github.com/open-source-parsers/jsoncpp. Accessed: 2019-01-09.

[4] Keystone: Open-source Secure Hardware Enclave. https://keystone-enclave.org. Accessed: 2019-01-09.

[5] Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/. Accessed: 2018-12-06.

[6] memaslap: Load Testing and Benchmarking a Server. http://docs.libmemcached.org/bin/memaslap.html. Accessed: 2018-12-06.

[7] memtier benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached. https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/. Accessed: 2018-12-06.

[8] Redis In-Memory Data Structure Store. https://redis.io/. Accessed: 2018-12-06.

[9] Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html. Accessed: 2018-12-31.

[10] SQLite Memory-Mapped I/O. https://www.sqlite.org/mmap.html. Accessed: 2018-12-06.

[11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[12] T. Ahonen, A. Hadid, and M. Pietikainen. Face Description With Local Binary Patterns: Application to Face Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):2037–2041, 2006.

[13] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[14] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-It-Yourself Virtual Memory Translation. In *44th Annual International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2017.

[15] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security. *ARM White Paper*, 3(4):18–24, 2004.

[16] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, volume 13, 2013.

[17] L. O. Andersen. *Program Analysis and Specialization for The C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, 2016.

[19] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992.

[20] A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 132–137. ACM, 2017.

[21] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3), 2015.

[22] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A.-R. Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv preprint arXiv:1709.09917*, 2017.

[23] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, 2016.

[24] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[25] L. Bryant, J. Van, B. Riedel, R. W. Gardner, J. C. Bejar, J. Hover, B. Tovar, K. Hurtado, and D. Thain. VC3: A Virtual Cluster Service for Community Computation. In *Proceedings of the Practice and Experience on Advanced Research Computing, (PEARC)*, pages 30:1–30:8, 2018.

[26] Z. Chang, D. Xie, and F. Li. Oblivious RAM: A Dissection and Experimental Evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 9(12):1113–1124, 2016.

[27] J. Chu and V. Kashyap. Transmission of IP over InfiniBand (IPoIB). *RFC*, 4391:1–21, 2006.

[28] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium*, pages 857–874, 2016.

[29] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–96, 2014.

[30] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *27th USENIX Security Symposium (USENIX Security)*, pages 1441–1458, 2018.

[31] G. J. Duck, R. H. C. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *24th Annual Network and Distributed System Security Symposium, (NDSS)*, 2017.

[32] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-time/Run-time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 186–197. ACM, 1996.

[33] S. Eskandarian and M. Zaharia. ObliDB: Oblivious Query Processing using Hardware Enclaves. *arXiv preprint arXiv:1710.00458*, 2017.

[34] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware From Software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 287–305, 2017.

[35] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[36] B. Fleisch and G. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 211–223. ACM, 1989.

[37] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. J. Patel, and W. mei W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, 2010.

[38] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[39] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667. USENIX Association, 2017.

[40] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016.

[41] S. M. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 1999.

[42] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. 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 (ASPLOS)*, pages 265–278, 2013.

[43] Intel 64 and IA-32 Architectures. Software Developer's Manual. *Intel Corp.*

[44] S. Jennings. Transparent Memory Compression in Linux. LinuxCon, 2013.

[45] Joshua Lind and Oded Naor and Ittay Eyal and Florian Kelbert and Peter R. Pietzuch and Emin Gün Sirer. Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR)*, pages 125–125, 2018.

[46] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–132, 1994.

[47] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pages 205–221, 2017.

[48] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pages 75–, 2004.

[49] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 285–298. USENIX Association, 2017.

[50] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 87–101, 2015.

[51] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[52] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *4th Workshop On Binary Translation (WBT)*, 2002.

[53] E. P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 177–190. USENIX Association, 1996.

[54] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. V. Rozas. Intel®Software Guard Extensions (Intel®SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, pages 10:1–10:9, 2016.

[55] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[56] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 291–305. USENIX Association, 2015.

[57] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003.

[58] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium*, pages 619–636, 2016.

[59] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. 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):28, 2018.

[60] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX Annual Technical Conference (USENIX ATC))*, pages 227–240, 2018.

[61] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pages 238–253. ACM, 2017.

[62] M. Owens and G. Allen. *SQLite*. Springer, 2010.

[63] P. J. Phillips, H. Wechsler, J. Huang, and P. J. Rauss. The FERET Database and Evaluation Procedure for Face-Recognition Algorithms. *Image Vision Comput.*, 16(5):295–306, 1998.

[64] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[65] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 431–446. USENIX Association, 2015.

[66] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24. IEEE, 2007.

[67] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[68] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th International Conference on Supercomputing*, pages 245–252. ACM, 1997.

[69] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP

Clusters. In *4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–136. IEEE, 1998.

[70] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *ACM SIGPLAN Notices*, volume 29, pages 297–306. ACM, 1994.

[71] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 36–47, 2003.

[72] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[73] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[74] S. Shahar, S. Bergman, and M. Silberstein. ActivePointers: A Case for Software Address Translation on GPUs. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 596–608, 2016.

[75] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87. USENIX Association, 2018.

[76] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[77] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[78] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, pages 299–310. ACM, 2013.

[79] Y. Sui and J. Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[80] Y. Sui, D. Ye, and J. Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[81] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 645–658, 2017.

[82] 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 *27th USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.

[83] R. A. Van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer, 2001.

[84] X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

[85] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 81–93. ACM, 2017.

[86] B. C. Xing, M. Shanahan, and R. Leslie-Hurd. Intel®Software Guard Extensions (Intel®SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, pages 11:1–11:9, 2016.

[87] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[88] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–298, 2017.

Secured Routines: Language-based Construction of Trusted Execution Environments

Adrien Ghosn James R. Larus Edouard Bugnion
EPFL, Switzerland

Abstract

Trusted Execution Environments (TEEs), such as Intel SGX enclaves, use hardware to ensure the confidentiality and integrity of operations on sensitive data. While the technology is available on many processors, the complexity of its programming model and its performance overhead have limited adoption. TEEs provide a new and valuable hardware functionality that has no obvious analogue in programming languages, which means that developers must manually partition their application into trusted and untrusted components.

This paper describes an approach that fully integrates trusted execution into a language. We extend the Go language to allow a programmer to execute a `goroutine` within an enclave, to use low-overhead channels to communicate between the trusted and untrusted environments, and to rely on a compiler to automatically extract the secure code and data. Our prototype compiler and runtime, GOTEE, is a backward-compatible fork of the Go compiler.

The evaluation shows that our compiler-driven code and data partitioning efficiently executes both microbenchmarks and applications. On the former, GOTEE achieves a 5.2× throughput and a 2.3× latency improvement over the Intel SGX SDK. Our case studies, a Go `ssh` server, the Go `tls` package, and a secured keystore inspired by the go-ethereum project, demonstrate that minor source-code modifications suffice to provide confidentiality and integrity guarantees with only moderate performance overheads.

1 Introduction

Our era is defined by the emergence of a digital society in which established notions of privacy, confidentiality, and trust are undercut by the shortcomings of today's technology, which is increasingly reliant on cloud computing. In the cloud, developers and users implicitly trust the cloud provider but are still susceptible to: (1) hardware and firmware flaws, such as the recent Meltdown [41] and Spectre [36] attacks, (2) vulnerabilities within the hypervisor [3], (3) exploits in libraries and Software as a Service (SaaS) infrastructures [1,2,4,5], (4) malicious employees with physical and administrative access to both computer and storage resources, and (5) intrusive or extra-territorial government surveillance [10,12,18].

To address these concerns, processor vendors, following ARM's lead [9], introduced Trusted Execution Environments (TEEs), a hardware mechanism based on memory encryption and attestation that isolates program execution and state from the underlying operating system, hypervisor, firmware, I/O devices, and even people with physical access to a machine. TEEs have been portrayed as *the* solution to the problem of trust in the cloud [16,17,39,51]. In particular, Intel SGX [6] partitions hardware and software into two mutually distrustful domains: a CPU, trusted user code, and a specified region of memory form the *trusted* domain, while the remainder of the hardware and software form the *untrusted* domain. SGX *enclaves* execute trusted user code in a trusted domain. Entering an enclave guarantees, through hardware, the confidentiality and integrity of the enclave's code, data, and execution.

Despite SGX's availability on current-generation processors, uptake has been slow, probably due to the absence of support on server-grade CPUs, the difficulty of programming enclaves, their performance overhead, and the need to refactor applications. The private messaging application Signal [43] is one of the few applications that appears to use enclaves, and Microsoft Azure only recently offered the first cloud solution to expose SGX features [42,45]. A major challenge is that this new technology lacks a clear programming model. Previous solutions fall into two broad categories: (1) run complete user applications in the trusted domain [16,17] and (2) separate the portions of a program that require trusted execution [7,8,40]. Solutions in the first category provide an abstraction, such as an operating system [17] or a container [16], to execute unmodified applications in an enclave. The other alternative requires a developer to identify and partition [7,8,46], or provide annotations that a program analysis tool can use to partition [40], an application into trusted and untrusted components. None of these prior approaches integrates the TEE into language-specific abstractions and semantics.

This paper describes an approach that fully integrates trusted execution into a modern programming language in an appropriate manner. We extend the Go language to allow a programmer to execute a `goroutine` within an enclave, to use low-overhead channels to communicate between the trusted and untrusted environments, and to rely on the compiler to automatically extract the code and data necessary to run the enclave. Our solution provides language support for trusted execution that is idiomatically compatible with the Go programming language.

We introduce *secured routines*, a new language-based feature that hides the hardware intricacies with little overhead. A secured routine is a user-level thread that executes a closure, *i.e.,* a function call, in the enclave at the request of untrusted code. The secured routine abstraction cleanly distinguishes trusted and untrusted code. Communications between the two domains are possible solely via *cross-domain channels*, an extension to native Go channels that deep-copies values to prevent cross-domain pointer references.

GOTEE extends the Go programming language with a single keyword, `gosecure`, to identify secured routines. GOTEE is an open-source fork of `golang/go` [15]. Starting from `gosecure` calls, the compiler identifies the minimal code required within the enclave and extracts it into a statically-linked trusted binary. Trusted and untrusted domains have their own runtime, memory management, and scheduler. GOTEE coordinates interactions between trusted and untrusted code, replaces control transfers between these domains with inexpensive synchronized data transfers using strongly-typed cross-domain channels.

Our contributions include:

- A language-based, expressive, strongly-typed, high-performance, remote-execution model for TEEs that strengthens isolation between trusted and untrusted code.
- A practical implementation of these ideas using the Go programming language and runtime. Our evaluation using microbenchmarks demonstrates that an enclave core serving secured routines can achieve 5.2× the throughput of domain-crossing control transfers.
- A demonstration that secured routines provide an expressive model to implement secured applications: we partitioned the `tls` module (a built-in Go library), protected a full ssh server, and extended the go-ethereum keystore (a popular cryptocurrency client) to isolate all operations that access private keys and certificates without a significant loss of performance.

We describe the necessary background (§2), the secured routine abstraction (§3), the implementation of GOTEE (§4), and evaluate it using both microbenchmarks and three security-sensitive applications (§5). Finally, we discuss possible architectural improvements in §6, related work in §7, and conclude in §8.

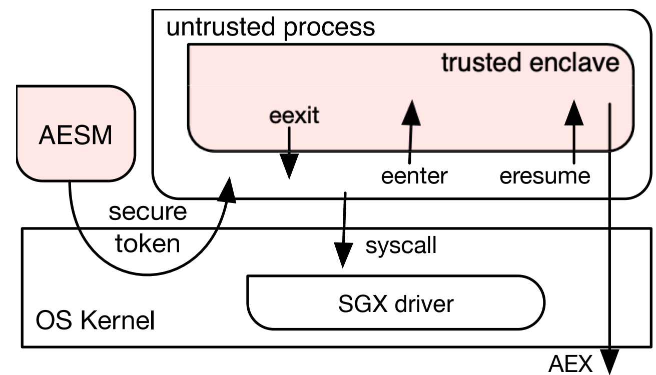


Figure 1: Trusted Execution Environments with Intel SGX; the enclaves and trusted parts are colored.

2 Background

2.1 Intel Software Guard Extension

Intel Software Guard Extension (SGX) [6], introduced in 2015 with Intel's sixth generation Skylake processor, allows user-level creation of *enclaves*. These are contiguous regions of virtual memory, protected against outside access and modification, even by software running at high privilege levels or by I/O devices.

Figure 1 illustrates the partition of a process between secure, trusted code and data and non-secure, untrusted code and data. SGX enforces an asymmetric trust model: the enclave has access to the entire memory, while untrusted code is unable to access or modify enclave memory. SGX further ensures that control from the untrusted domain enters the enclave only at pre-approved entry points. In SGX, Intel provides the root of trust through the `aesm` module that cryptographically ensures the validity of the initial state of the enclave.

SGX reserves, at boot time, a contiguous portion of physical memory, called the *Processor Reserved Memory* (*PRM*), with a maximal size of 128 MB. A 94 MB subset of this region, called the *Enclave Page Cache* (*EPC*), is used to allocate enclave memory pages. The integrity of the EPC is ensured through Merkle trees implemented in hardware [11]. The EPC size is a hard limitation on the amount of code and data that can be loaded into an enclave without incurring expensive page evictions to regular DRAM [17, 53]. The CPU's Memory Management Engine (MME) ensures confidentiality by encrypting cache lines evicted to memory and by decrypting them as they are brought into the CPU running in enclave mode; this reduces the available memory bandwidth by 4× [53].

Creation of an enclave requires the execution of a complex instruction sequence using new instructions such as `ecreate`, `eadd`, `eextend`, and `einit` that respectively create the enclave; define its resources together with their initial state and

access rights; and finally initialize the enclave. The number of concurrent threads allowed inside the enclave corresponds to the number of `eadd`ed Thread Control Structures (*TCS*) and is fixed at enclave initialization.

After enclave initialization, user-level software uses the `eenter` instruction to perform an `ecall`, a control transfer to a pre-defined location within the enclave (Figure 1). The `eexit` instruction allows to perform `ocalls`, *i.e.,* a voluntary control transfer to untrusted code. SGX also supports asynchronous enclave exits (AEX) to service interrupts and exceptions, which is necessary since the enclave forbids privileged instructions. An AEX saves the current state of the enclave within the EPC, restores the untrusted context, and transfers control to the operating system handler. The untrusted code resumes enclave execution by performing an `eresume`.

Finally, SGX provides a remote attestation mechanism that allows developers to verify the integrity of the software in the enclave. As part of enclave creation, developers need to provide a *measurement* of the enclave, *i.e.,* a signed hash of the SGX instructions and arguments used to instantiate the enclave, as well as of selected portions of the enclave's code and data. A remote party can compare this measurement with its expected, precomputed value and proceed with the enclave's execution only if the two values match.

2.2 Building Secured Systems

One approach to utilizing SGX is to run all of an application in the enclave. The literature contains examples of complex abstractions—including an entire operating system, Haven [17]; a library operating system, Graphene [23]; and a container platform, SCONE [16]—running in SGX. While convenient for developers and effective at reducing expensive enclave crossings [53], this approach has significant drawbacks: (1) it greatly expands the amount of code running inside the enclave, which puts pressure on system resources and incurs pervasive memory decryption overheads and (2) it brings into the enclave code and third-party libraries—not necessarily used, understood, or validated—which can facilitate attacks on the enclave (*e.g.,* ROP [47]).

Another approach necessitates a deeper understanding of an application, as it requires splitting the application's code and data into trusted and untrusted portions, following the *Intel SGX Software Development Kit* (*SDK*) [7] model. This SDK is a set of C/C++ libraries and tools that enable programmers to create and deploy enclaves. The Intel SGX SDK exposes an API similar to the SGX instructions. Trusted and untrusted code and data reside within distinct source files. A configuration file describes the required *ecall*s and *ocall*s functions. The compilation process first invokes an IDL compiler to generate boilerplate code and then compiles the enclave code as a position-independent binary with all dependencies statically linked, invokes a signing tool on this `.so` to meter the enclave's code, and finally compiles the untrusted application code as a regular executable.

SCONE [16] and Eleos [44] both rely on message passing to implement asynchronous system calls and avoid expensive enclave exits. Following the same approach, Intel recently published `switchless` [14], a (under-development) mini-framework that provides a simple C++ messaging mechanism on top of the SDK.

Asylo [31] is a C++ framework, compatible with gRPC [32], that abstracts TEE technologies behind a concise API and a set of C++ classes. From a practical point-of-view, Asylo is an improved version of Intel's SDK that exposes a smaller API, requires less boilerplate code, supports different TEE implementations, and provides transparent support to perform system calls from the enclave.

Glamdring [40] automates code partitioning, as it only requires a developer to mark data that needs to be protected. It then relies on static analysis to determine the portion of code that accesses this data and needs to run within the enclave. As an optimization, Glamdring uses heuristics to enlarge the trusted code base and limit the number of expensive enclave crossings. This can help balance a trade-off between EPC memory consumption and the number of domain crossings. Glamdring provides less fine-grained control over code partitioning than the Intel SGX SDK, but hides the technology's intricacies and exposes a very simple programming model.

2.3 SGX Limitations

The SGX technology, and its implementation on current Skylake processors, presents major performance challenges: while the magnitude of these overheads may change in the future with refinement of the processor's micro-architecture, or by adding dedicated silicon, these overheads are, to some extent, tied to the mechanisms providing confidentiality and integrity in the SGX design.

The limited EPC working set and the reduced memory bandwidth are inherent in the design. Similarly, the control-flow transitions between the trusted and untrusted execution (*i.e.,* ecalls, ocalls, and AEX) are expensive because of TLB shootdowns, CPU state changes, and cache flushes needed to mitigate foreshadow attacks [22]. These domain crossings are an order of magnitude more expensive than a system call [53], between $\sim$2µs [53] and $\sim$3.5µs on our hardware. This corresponds to a throughput of less than 1M enclave entries per second with four cores performing `ecalls` in parallel (see §5.2). Keep in mind that system calls within an enclave require a domain crossing, as SGX is limited to user-level execution, and as a consequence these calls also become an order of magnitude more expensive.

Put together, these limitations require a programmer to worry about the size of the trusted code base and the trusted working set, to reduce the exposed attack surface as well as the frequency of EPC page evictions; to optimize the application for cache locality; to understand precisely the threading model

of the application; and to minimize domain crossings, system calls, interrupts, and signals.

3 Design

The *secured routine* extension to Go enables GOTEE to partition an application's code, data, and execution between trusted and untrusted domains, while *cross-domain channels* reinforce memory isolation and enable cross-domain communication and cooperation. This section presents a high-level description of the design and semantics of GOTEE's extensions.

3.1 Threat Model

We follow the threat model of other work in SGX [16, 17, 23, 40] in which an adversary tries to access confidential data or to damage the SGX enclave's integrity. The attacker has administrative access to the machine and control over both hardware and software, and may modify any code or data in untrusted memory, including the operating system and the hypervisor. We consider Iago attacks [24] for GOTEE's runtime and system call interposition mechanism in Section 4.3.

Denial-of-service attacks, a known limitation of SGX [26], and hardware side channels (*e.g.,* based on caches, page faults, or branch shadowing) are out of scope. We assume a correct underlying implementation of SGX that provides confidentiality and integrity for enclave code and data.

3.2 Quick Overview of `Golang`

The Go programming language (`golang`) is a modern, memory-safe, garbage-collected, structurally-typed, compiled, systems programming language. Go supports concurrency based on the Communicating Sequential Processes (CSP) model [33]. The unit of execution within a Go program is called a *goroutine*, a user-level thread executing a closure that is created by prefixing a function call with the `go` keyword. Goroutines are multiplexed and scheduled on a pool of operating system threads, using a cooperative scheduling model implemented by the Go runtime. Goroutines communicate and synchronize using *channels*, which are synchronized, typed message queues with copy and blocking read/write semantics.

3.3 Secured Routines & Cross-domain Channels

From a programming point of view, a secured routine provides a simple and familiar abstraction that allows a programmer to execute a goroutine within an enclave and to use cross-domain channels to communicate between the trusted and untrusted environments.

```
1 var secretKey *Key
2 func generateSymKey(*io.Reader) *Key {...}
3
4 func InitSymKey(done chan bool){
5     fmt.Println(``Creating a new secret key``)
6     secretKey = generateSymKey(rand.Reader)
7     done <- true
8 }
9
10 func EncryptServer(request, reply chan []byte){
11     for {
12       msg := <- request
13       reply <- secretKey.Encrypt(msg)
14     }
15 }
16
17 func TrustedEncryption(msg []byte) []byte{
18     done := make(chan bool)
19     gosecure InitSymKey(done)
20     _ = <- done
21     request := make(chan []byte)
22     reply := make(chan []byte)
23     msg := []byte(``The quick brown fox...``)
24     gosecure EncryptServer(request,reply)
25     request <- msg
26     res := <- reply
27     fmt.Println(``Encryption done``)
28     return res
29 }
```

Listing 1: Using secured routines to isolate a secret key within the TEE.

Listing 1 presents a sample program that secures a secret encryption key, `secretKey`, within the enclave. The `TrustedEncryption` function uses the `gosecure` keyword to spawn a secured routine that creates the key within the enclave. A subsequent `gosecure` call spawns an encryption server, `EncryptServer`, within the enclave. The untrusted code sends the message to the server (line 25) and gets back the encrypted result (line 26).

The programmer relies on `gosecure` to inform the compiler how to partition the code between trusted and untrusted domains. The compiler determines the functions that can be reached by the execution within the enclave, in this example `InitSymKey`, `EncryptServer` and their dependencies `fmt.Println`, `generateSymKey`, `*Key.Encrypt`, *etc.* GOTEE compiles these functions into a statically-linked executable.

Unlike prior work [7, 8, 40], secured routine's code partitioning does not require disjoint trusted and untrusted code. Functions can exist in both environments, *e.g.,* the function `fmt.Println` in Listing 1.

GOTEE hardens memory isolation between trusted and untrusted domains, as compared with the SGX hardware model, in three ways. First, each domain manages its own set of symbols, data, and global variables independently, allowing them to have distinct copies of data and globals. This also differs from Glamdring [40], where the trusted and untrusted

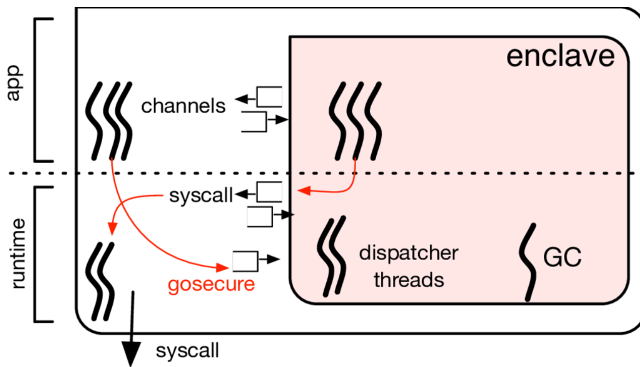


Figure 2: Channel-based cooperation between runtimes.

namespaces cannot overlap.

Second, GOTEE allows only *cross-domain channel*s across the trusted boundary. Cross-domain channels are an extension to the native Go channels allowing secured communications across domains with *deep-copy* semantics. Cross-domain channels are declared and used like regular go channels. However, they provide deep-copy semantics to prevent pointers from crossing a domain boundary. For example, in Listing 1, the `msg` byte slice received at line 12 is an in-enclave copy of the untrusted one sent at line 25.

Third, function arguments passed to secured routines, with the exception of cross-domain channels, are deep-copied inside the enclave by GOTEE's runtime. The deep-copy mechanism can be seen as a marshalling step similar to the one needed to send complex objects or structures over a network. GOTEE emits compilation warnings if a deep-copy, due to a secured routine or a cross-domain channel, requires to dereference a pointer.

While more restrictive than the original SGX model, GOTEE's design ensures that enclave code cannot be subverted or leak secrets by inadvertently dereferencing or writing to an unsafe memory location. All data that leaves the enclave does so by being explicitly sent over a cross-domain channel, while all data referenced by the application's trusted code resides in the enclave.

3.4 Runtime Cooperation

The secured routine abstraction requires mutually distrustful domains to cooperate. More specifically, it allows the untrusted domain to trigger execution of a closure within the trusted one. For example, when the untrusted execution reaches line 19 in Listing 1, the trusted runtime spawns a new routine that invokes the `InitSymKey(done)` closure.

Figure 2 presents the general overview of runtime cooperation. Both domains have their own code and data, their own thread pools to multiplex execution, and their own managed memory regions that are separately garbage collected. Between the two domains, dedicated cross-domain channels are used by the runtimes to trigger the execution of secured routines and to enable enclave system calls. Specifically, and unlike a normal `go` closure, a secured routine is implemented by passing its arguments on a dedicated channel not visible to `golang` programmers. The trusted runtime verifies the validity of the closure's entry point before scheduling it within the enclave. System call interposition operates in a similar manner: the trusted runtime copies the system call's arguments into a dedicated, hidden channel; the untrusted runtime then reissues the system call asynchronously and returns the result over a private channel.

Since full copy semantics are enforced between the two domains, each garbage collector can safely manage its own memory space without synchronizing with the other one.

3.5 Compatibility With SGX

The secured routine abstraction and its design are compatible with the SGX technology and its performance model:

Minimum trusted code: The code loaded into the trusted domain is automatically extracted by the compiler and is minimal. This is both security- and resource-efficient as it reduces the number of EPC pages consumed by the enclave as well as its attack surface.

Control transfers: Control transfers between the two domains are replaced with inexpensive, synchronized, and typed data transfers via cross-domain channels for both application-level communication as well as runtime synchronization. The expensive SGX domain crossings are only necessary in the initialization phase, to block threads when they are idle or in the stop-the-world GC phase, and to service an EPC miss.

Defensive programming: Cross-domain channels, used to launch closures and to invoke system calls, perform memory copies and sanitize arguments. Moreover, they are the single point of interaction between mutually distrustful domains and are therefore easy to augment with defensive programming techniques.

Thread multiplexing: The SGX environment chooses, at enclave creation time, the number of threads that can execute simultaneously within the trusted domain. The Go thread pool size can be fixed at the beginning of the execution to match the number of TCS in the enclave. This, however, does not impose any limitation on the number of concurrently executing secured routines, which means that concurrency is not bounded by this SGX limitation.

System call interposition: The use of channels to communicate and synchronize between the two runtimes simplifies system call interposition. The runtime detects system calls from trusted code, performs argument sanitization, copies arguments to untrusted memory buffers, and sends the system call to the untrusted runtime. Once the system call is serviced, the enclave runtime can perform additional checks to validate the result before delivering it to the application.

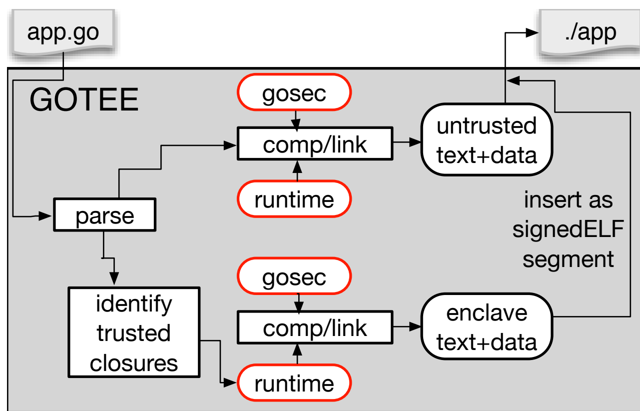


Figure 3: The GOTEE compilation pipeline.

No global variables or cross-domain references: secured routines reinforce the isolation between the two domains by prohibiting shared global variables and cross domain memory references. This forces data sharing to be explicit and passed through either typed communication channels or typed function arguments, with deep-copy semantics. This design eliminates implicit sharing and cross-domain references, which pose the risk of mistakenly leaking data and violating confidentiality.

Secured routines do not provide any guarantee or protection with regards to denial of service attacks. As with previous work [16,23,40], we consider the challenge of bringing secrets into the enclave to be out-of-scope for this paper. These are known, fundamental limitations of the SGX technology that GOTEE does not ameliorate.

Compatibility with other TEE designs: The secured routine abstraction is not tied to the SGX model. From a high-level point-of-view, secured routines and cross-domain channels allow cooperation between two (memory-isolated) peer environments that communicate solely via specific channels. The GOTEE compiler can be extended to support other TEE implementations without requiring application code modifications.

4 Implementation

The GOTEE compiler and runtime extend the Go system. This section describes the changes to the compiler, a new library written in Go that provides SGX support, and the changes to the runtime environment.

4.1 Compiler Support for `gosecure`

The GOTEE compiler is responsible for partitioning code and data according to the design of §3.3. GOTEE is a backward-compatible extension of the standard Go compiler with a new keyword `gosecure`, and an extension to Go channels, *cross-domain channels*. The changes are small, consisting of ∼400 modified lines and ∼2000 lines of new code written in pure Go.

Figure 3 illustrates the process: GOTEE compiles each instance of `gosecure` by type-checking and validating the closures at compile time. The generated code differs slightly from the standard goroutine support. On the caller side, the closure arguments are sent over a cross-domain channel. On the callee side, within the enclave, the runtime library pulls the in-enclave copy of the closure arguments and a function identifier from the channel, validates the target function, spawns the corresponding routine with the arguments, and then schedules it. Compared to a standard goroutine, GOTEE adds a level of indirection, with a write to and read from a cross-domain channel, and the deep-copy of each argument.

GOTEE records functions with the `gosecure` keyword as valid targets for the secured routine abstraction within the enclave. GOTEE then initiates a full compilation for enclave code, using the Go compiler's analysis to determine the minimum transitive closure of code reachable from these functions, as well as the global variables used by this code. The compiler also creates a `main` function for the enclave that serves as the `eenter` entry point and that initializes the runtime servers for cross-domain cooperation. The result of this compilation step is a statically-linked, non-relocatable binary to be loaded into the enclave as the trusted code.

GOTEE implements restrictions on the enclave code. First, the compiler detects channels passed via arguments to secured routines and ensures that these are declared as cross-domain channels. Second, the compiler inspects secured routine's target signatures as well as cross-domain channel types and emits warnings if their deep-copying requires dereferencing pointers. Third, GOTEE does not allow function pointers as arguments to secured routines or cross-domain channels. Finally, GOTEE only allows pure Go code within the enclave and rejects dependencies on C code and shared libraries.

GOTEE also compiles the untrusted code using the standard Go compiler, without these restrictions. As a final step, GOTEE packages the statically linked trusted executable into an ELF segment of the untrusted binary.

GOTEE can optionally generate a signed measurement of the enclave at compile time and store it within a dedicated ELF section of the untrusted binary, so as to perform remote attestation upon deployment. If not done by the compiler, the measurement and signature of the trusted code can be performed at run time.

4.2 `gosec` – an SGX Library in Go

The GOTEE compiler includes an SGX library, completely implemented in Go, as a standard Go package called `gosec`. It contains ∼1000 lines of code.

Loading an enclave: `gosec` mirrors the Intel SGX API in that it provides functions to (1) create an enclave, (2) load

a static binary into the enclave, (3) take a measurement of the enclave, and (4) perform `eenter` and `eresume` to the enclave. The `gosec` package communicates with the Intel SGX Linux kernel driver via `ioctl` to execute the privileged SGX instructions, *i.e.,* `ecreate`, `eadd`, and `einit`. It also communicates with the Intel `aes` module [26] that delivers the token required to perform the initialization (`einit`, see §2.1). The `gosec` package implements step (2) by parsing the ELF binary and extracting the enclave code. At run time, the package spawns a new, untrusted, operating system thread to execute an `eenter` instruction that starts the enclave. The number of concurrent threads allowed inside the enclave can be selected by setting an environment variable. By default, the loader adds only two TCS to the enclave: one to execute the user code, the other to support garbage collection.

Measuring an enclave: Measuring an enclave is a series of distinct steps that involve the SGX driver (to execute privileged instructions), the SGX daemon (to retrieve a cryptographic token), the measurement byte array generated by the `gosec` library while creating the enclave [26] (§4.1), and the enclave binary itself. First, the enclave's memory boundaries are determined by reading the ELF sections of the trusted binary. This information is used to perform the `ecreate` call. Then, individual page contents are registered via the driver, which performs the `eadd` and `eextend` accordingly. At the same time, `gosec` builds the corresponding measurement byte array, which is then used to retrieve a token from the SGX `aes` module daemon. Finally, `gosec` issues the `einit` driver call, using the token, to finalize the enclave.

AEX handler: Asynchronous exits from the enclave, *e.g.,* faults and exceptions, are first passed to the operating system. Then, a user-space AEX handler, implemented in `gosec`, is called. The handler runs outside of the enclave and plays a fundamental role in the debugging process. The `gosec` AEX handler reads a shared region of memory where the GOTEE runtime dumps information before performing a panic or throwing an exception. This, of course, is reliable only for debugging purposes. If no GOTEE runtime cause for the AEX is found, the `gosec` AEX handler performs an `eresume` to return in the enclave.

4.3 GOTEE Runtime

The third component of GOTEE is the runtime library that is statically linked to the enclave code. It consists of the Go runtime modified to run in an enclave, including its cooperative user-level thread scheduler and garbage collector, and extensions to allow trusted and untrusted code to cooperate. It supports cross-domain channels as the *sole* means of communications with untrusted code. The code patch consists of ~760 lines of new code and ~300 modified ones.

Enclave runtime initialization: GOTEE replaces most of the Go runtime initialization steps. The `gosec` package preallocates all trusted heap, thread local storage, and memory pools during the enclave creation as part of the load and initialization sequence. This is necessary because of the SGX metering requirements. As a result, the entry point of the enclave simply switches execution onto a protected stack that is part of the enclave and skips over most of the Go runtime memory allocation steps. After this, the enclave runtime shares most of the Go runtime, with minor changes to avoid enclave-disallowed instructions such as `cpuid` or `rdtsc`.

Allowing multiple trusted threads: GOTEE lazily spawns enclave threads. During the execution, when a new thread is required, the current enclave thread first atomically acquires a TCS from the pool. It then performs an enclave exit and a clone system call before resuming its enclave execution. While exits and entries are expensive, these are bounded by the maximal number of TCS allocated for the enclave. The newly created thread performs an `eenter` and jumps to the pre-defined enclave entry point to initialize its state before serving secured routines.

Securing untrusted channels: The channel implementation, as well as the goroutine structure, were extended to support a secured communication mechanism between the trusted and untrusted environments. To pass copies of values to and from secured routines, GOTEE uses buffers allocated within the unprotected memory region. Upon performing a blocking operation, the trusted runtime allocates an unprotected buffer that will either hold the value that it writes, thereby allowing an untrusted routine to access it, or be used to receive the value produced by an untrusted routine's write to the channel. When unblocked, the secured routine copies the content of the buffer to the appropriate memory location within the enclave. For complex types, the enclave performs a deep-copy. This adds an extra step for secured routines compared to standard Go, which allows direct read/write to the blocked goroutine's enqueued address, *e.g.,* a stack, a heap, or a data variable. GOTEE automatically identifies and instruments cross-domain channels at runtime, hence limiting the effort required to port existing applications. Communications within the same domain are unaffected.

Cross-domain synchronization: The two runtimes, and in particular their schedulers, must cooperate to synchronize access to channels across domains to ensure the timely delivery of messages. In Go, a blocking operation on a channel deschedules the routine and wraps it within a special datastructure along with a pointer to the read (write) memory location. In the case of a cross-domain channel, the wrapper must be accessible from both runtimes. GOTEE's enclave runtime manages a private untrusted memory area from which such wrappers are allocated. A secured routine that needs to enqueue itself will therefore allocate a wrapper, along with an untrusted memory buffer, and then enqueue itself in the untrusted cross-domain channel.

The unblocking operations on cross-domain channels

also required changes. An untrusted routine cannot directly reschedule a trusted routine, and vice versa. Instead, unblocking a routine enqueues it in a ready queue that belongs to the appropriate domain. These queues are polled by the corresponding runtime's scheduler. The scheduler ensures that the address of the goroutine is valid, *i.e.,* that it was registered at creation and is still live, before executing it. Note that this extra step only applies to cross-domain communications.

Memory management: The GOTEE runtime restricts the amount of available heap memory because of SGX memory-size limitations. The standard Go runtime assumes a 64-bit address space with gigabytes of memory and places its runtime heap, spans, and bitmap for memory management accordingly. During runtime initialization, and throughout code execution, the Go runtime `mmaps` portions of the address space corresponding to these regions and frequently extends them. An enclave's maximum memory live working-set is 94 MB, and even less if we want to avoid page evictions. As a result, GOTEE uses a fixed-size heap whose address and size are computed as a fixed offset from the code and data. The heap size can be set either at compile time if a measurement is generated or at run time before loading the enclave.

Thread Local Storage: Go relies on thread local storage (*TLS*) to quickly access runtime values such as the current routine (*G*) or the current machine abstraction (*M*). Go normally allocates *M* in the heap and sets it as the TLS base. SGX, on the other hand, requires a TCS to declare its TLS at creation time. GOTEE circumvents the SGX limitations by preallocating *M*s into the enclave's `.bss` segment. As all `.bss` data structures are part of the garbage collector's root set (unlike an arbitrary location in memory), this approach allows the enclave to use the unmodified Go garbage collector.

Garbage collection and Stack shrinking: Go performs mark and sweep concurrent garbage collection. The GC requires a short pause time with all threads blocked at a safe point for mark and sweep terminations. As a result, secured routines need a way to exit the enclave and perform a blocking `futex` sleep. Other than that, the original Go GC is unmodified, and it executes independently from the untrusted domain's runtime. Untrusted memory buffers are allocated and managed by an in-enclave allocation library and are not traced by either GCs. The trusted runtime keeps references to secured routines blocked on cross-domain channels, which both allows a safety check when they are rescheduled and keeps them in the live-set of objects during garbage collection.

Goroutine stacks can shrink and stack frames can be relocated in memory when the goroutine is blocked on a channel. In standard Go, the destination location of channel data may be on the stack, and therefore handled as part of stack relocation. In GOTEE, when a secured routine is blocked on a cross-domain channel, the destination address points to a location in untrusted memory, *i.e.,* not on the stack, while the stack pointer used as the final recipient of the deep-copy is the one updated during stack shrinking.

Mitigating SGX limitations: The current version of SGX disallows several instructions in the enclave, such as `syscall`, `cpuid`, and `rdtsc`. While these have to be completely avoided during the runtime initialization, due to the limited environment at that time (no heap or channels during the early init phases), they can later be emulated. The system call interposition mechanism allows the enclave to forward system calls to the untrusted runtime. The same mechanism can be used to execute a `rdtsc`, with the communication overhead reducing its accuracy. For the `cpuid` call, most of the information provided by the instruction is fixed at enclave creation, which simplifies its emulation.

Go relies on `futex` calls to implement locking within the runtime. These are optimistic locks, performing a limited amount of spinning before sleeping. In an enclave, a `futex sleep` would require to exit the enclave and re-enter upon a `futex wake up`, with high overheads. Instead, in GOTEE, a secured routine that needs to obtain a cross-domain channel lock will spin until it acquires the lock. Upon an unlock, GOTEE checks if any unsafe thread is sleeping on the `futex`. If so, it spawns a dedicated routine to use the system call interposition to perform the unblocking `futex wake up` system call. This approach is similar to the one used by standard Go for blocking system calls, except that GOTEE relies on routines rather than operating system threads.

Network support: The Go runtime relies on `epoll` calls, as part of the scheduler's logic, for network events. GOTEE extends the scheduler's implementation to ensure that a single idle thread at a time is allowed to exit the enclave and perform the `epoll` call.

Iago Attacks: GOTEE's runtime is hardened against Iago attacks and only relies on 4 syscalls: `mmap` to allocate unsafe memory, checked against enclave boundaries and known unsafe areas; `futex` calls for idle threads, which are used to reduce CPU utilization, not mutual exclusion; and `epoll` calls performed by idle threads as described above.

On the application side, GOTEE provides a single point of system call interposition which relies on channels with deep-copy semantics for memory isolation. This currently performs system call filtering and safety checks on both arguments and results, and could be extended, in the future, to allow user-defined filtering policies.

Debugging: Debugging code within an enclave is challenging as the AEX user-space exception handler provides little information to identify the cause of an asynchronous exit from the enclave. GOTEE has an optional flag that allows a program to run in a simulation environment with identical memory layout and run time behavior as the SGX program, but without the SGX protection mechanisms.

	Workload	Text	Data	RO-data	Total	Main Package Dependencies	Application LOC
GOTEE	runtime only	493	25	273	793	-	-
	hello world	+72	+1	+16	+91	++ fmt, syscall, strconv, os, io, reflect, runtime, unicode	13
	enclave-cert	+174	+1	+45	+221	++ crypto/rsa, math, bytes, hash, unicode	75
	ssh	+1036	+4	+291	+1332	++ golang.org/x/crypto/*, crypto, gnet, encoding	71
	keystore	+1165	+ 4	+329	+1499	++ crypto/ecdsa, crypto/elliptic, crypto/aes	474
SDK	runtime only	67	2	4	75	-	-
	hello world	+49	+0	+1	+51	-	355

Table 1: Per case-study enclave TCB breakdown in KB, package dependencies, and application lines of code (LOC). **+** and **++** are, respectively, an increase over the baseline `runtime only`, and over all previous table entries.

5 Evaluation

Our experiments were performed on a Microsoft Azure Cloud Confidential Computing server, with an Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz with 4 physical cores, configured with Ubuntu 18.04 LTS running Linux kernel 4.15.0-1036-azure. GOTEE operates with the standard Intel SGX 2.0 Linux kernel driver (`sgx2`) and attestation daemon (`aesm`). All GOTEE experiments were run with garbage collection enabled and a single thread servicing secured routines in the enclave.

The purpose of our evaluation is to validate: (1) the effectiveness of secured routines as a way to partition code (§5.1); (2) the performance, latency, and throughput of secured routines and their cost in comparison to the crossing-oriented approach of the Intel SDK (§5.2); (3) with three case studies, GOTEE's usability and ability to hide critical secrets within the enclave by executing a full application in the enclave(§5.3), by performing a fine-grained partitioning of a standard Go package(§5.4), and by extending a real-world application with a TEE-specific implementation(§5.5).

5.1 Code Size

We first evaluate the impact of secured routines on the enclave code size. To this end, we add a baseline `hello world` benchmark that invokes `fmt.Println` in the enclave, and compare it to the Intel SDK C++ `hello world` code sample.

Table 1 shows, for each case study, (1) the size of the enclave code measured as an increase on the baseline size of GOTEE runtime for the enclave, (2) the main Go package dependencies, and (3) the application lines of code. Entries in the table are sorted such that each case study only reports extra packages imported compared to previous lines.

First, we observe that both the Go runtime and the generated code are larger than the C++. Second, the `ssh-server` is responsible for the greatest increase, in TCB size, over the runtime baseline, due to its numerous dependencies. This result is expected as this particular case study does not leverage the fine-grain partitioning provided by GOTEE and simply puts the entire application code inside the enclave. The `keystore` prototype only adds a few crypto subpackages to the TCB dependencies.

On the other hand, Table 1 also shows the difference in source-code level complexity between GOTEE and the Intel SDK. In `hello world`, the lack of transparent forwarding of system calls in the SDK requires a programmer to forgo `printf` in the enclave and instead: (1) call `sprintf` to write to an intermediate buffer, (2) define and `ocall` with the IDL compiler, and (3) use it to issue a `write` system call. Additionally, programmers are still responsible for properly implementing all the boilerplate code required to define, create, and load the enclave. As a result, the C++/SDK `hello world` consists of 355 LOC, 13 files, requires 85 lines of configuration, and 161 lines of `Makefile`.

By comparison, the GOTEE 13 lines of code `hello world` compiles with the `gotee build` command.

5.2 Microbenchmarks

This evaluation uses the following microbenchmarks:

- `syscall`-lat: from within a trusted closure, execute a `getuid()` system call in a loop; report the mean latency.
- `gosecure+block`-lat: spawn a trusted closure and wait for a response over a private cross-domain channel; report end-to-end median latency.
- `gosecure-server`-lat: a single secured routine performs blocking writes to a cross-domain channel in a loop. An untrusted routine measures the latency of performing a read on the same channel. The difference between this measurement and `gosecure+block`-lat corresponds to the runtime overhead required to trigger a secured routine.
- `gosecure`-tput: multiple untrusted goroutines concurrently spawn a trusted closure and wait for a response over a private cross-domain channel.

bench-name	Go	GOTEE	SDK
syscall (getuid)	0.23	1.35	3.69
gosecure+block	0.30	1.5	3.50
gosecure-server	0.20	0.60	-

Table 2: Latency microbenchmarks in µs.

- gosecure-server-tput: a single trusted closure receives requests on a public cross-domain channel from multiple concurrent untrusted goroutines and replies individually on private channels, effectively bypassing the runtime cooperation required to spawn new secured routines.

GOTEE latencies: Table 2 compares the latencies of basic operations in Go, GOTEE, and, when applicable, the equivalent C++ implementation with the Intel SGX SDK. All experiments report the median (mean for syscall-lat) over 500K iterations.

The latency to spawn a secured routine and have it write to a private channel is 1.5µs. The equivalent standard Go program has a latency of 0.30µs, suggesting that GOTEE runtime cooperation and SGX memory overheads have an impact of ∼1.2µs (5.0×). We believe that the implementation can be optimized to reduce contention on cross domain events and runtime cooperation overheads. Still, GOTEE shows a 2.3× improvement over the Intel SDK latency, which requires a full crossing (eenter followed by eexit).

For a trivial system call, that requires going through the syscall interposition mechanism over channels, GOTEE is able to achieve a 2.7× improvement over the Intel SDK crossing-oriented approach.

GOTEE throughput: The throughput experiments consist of multiple concurrent requests to the enclave. For the Intel SDK, different threads perform *ecall*s in parallel, yielding a throughput of 281 Kops for one thread and 938 Kops for all four cores.

For GOTEE, Figure 4 presents two variants, running with a single thread inside the enclave: (1) gosecure-tput, the closest in behavior to the Intel SGX SDK, and (2) gosecure-server-tput. The former shows a throughput improvement of 5.2× (1.46 Mops) over the SDK for a single core running in the enclave. GOTEE can allow a single enclave thread to achieve 1.6× the throughput of four cores executing the Intel SDK. GOTEE's throughput depends on the number of concurrent untrusted goroutines (multiplexed on a single thread) performing gosecure calls. For fewer than three untrusted goroutines, the runtime cooperation requiring to reschedule the secured routines dispatcher is the main bottleneck. After that, there are enough concurrent goroutines to avoid blocking the dispatcher.

The second GOTEE experiment shows the benefit of avoiding the secured routine creation overheads. Its performance degrades, however, as contention on the cross-domain channel increases; both runtimes compete to obtain the lock and must cooperate to reschedule unblocked routines. Vanilla Go, which is not subjected to our cooperation overhead or SGX performance costs, achieves over 4.1Mops.

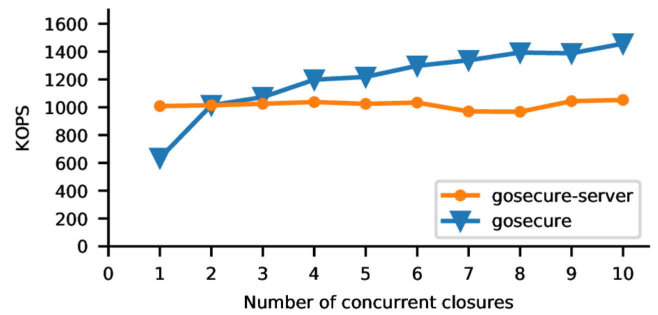


Figure 4: Synchronous closure execution rate for secured routine multiplexed on top of a single enclave thread.

The garbage collector's impact on these microbenchmarks is negligible. The Go memory statistics show that for the full throughput experiment, 21 GC cycles were completed inside the enclave, with a median pause time of 13µs. However, the total GC pause time only accounts for 0.033% of the application's available CPU time. In the latency microbenchmark, we measured a similar median for 2 completed cycles, which accounts for 0.015% of the benchmarks CPU time.

5.3 A full in-enclave ssh server

GOTEE can be used to port a full application to the enclave. The enclave size breakdown is reported in Table 1. The Go programming language provides, under golang.org/x/crypto/ssh, a fully functional implementation of an ssh-server. This implementation relies on the default net package. While *none* of the application logic code for the server was changed, this port required a few modifications to the net package, which relies on C bindings for socket structures in order to stay compatible with the Linux kernel headers. As GOTEE allows only pure Go code inside the enclave, we created gnet, a new package that redefines relevant C structures (*e.g.,* struct_sockaddr, struct_in_addr, struct_addrinfo) and constants in pure Go. This package adds 70 LOC to the native net package.

5.4 Webserver with enclave-cert

The loss or leakage of an SSL private certificate can have serious reputational consequences. However, a private certificate must reside in the memory of the process that handles an SSL connection. Our case study designs and implements the enclave-cert package, which isolates within an enclave the two operations that require access to an SSL certificate's private key: signing the handshake hash and decrypting the client's symmetric session key.

We modified the native Go `tls` package to allocate the server's private certificate key within the enclave and to perform these operations in the trusted environment. The `enclave-cert` package uses channels to pass encryption and decryption requests to the enclave. A single secured routine is spawned by the user application when a certificate is loaded or created. The secured routine then waits on the request channel, performs the requested decryption, and notifies the untrusted requester.

The code patch consists of 9 additional LOC that add optional request channels to the TLS certificate structure. The enclave code is in `enclave-cert`, a new package of 35 LOC that defines the operations on the private key. The `http` package is unmodified. Any webserver application that uses `enclave-cert` operates like a corresponding Go webserver application.

The separation of functionality between the `tls` package (which does not depend on GOTEE or `gosec`) and `enclave-cert` eliminates circular dependencies and ensures backward compatibility when SGX is not available.

In this experiment, we have an `apache-bench` client connect repeatedly over `https://localhost` to a simple webserver and request a single page load per session. The workload is totally dominated by the TLS handshakes.

We compare the built-in Go `http` and `tls` packages with the modified `enclave-cert`. The built-in server achieves an average of 400 reqs/sec, while `enclave-cert` achieves 353 reqs/sec (*i.e.,* 88% of native). The `apache-bench` output shows they have the same mean for connection and processing time, but `enclave-cert` has a higher (6×) standard deviation. In fact, the run time cooperation between trusted and untrusted domains is a source of variability that impacts the system's stability and tail latency. A similar experiment with Glamdring [40] reported only 60% of native throughput due to the cost of frequent enclave crossings.

5.5 Keystore based on go-ethereum

The go-ethereum [29] project is the official implementation of the Ethereum protocol [13] in Go. A particular feature of the project is the ability to manage ethereum signature keys (ECDSA) as part of a `keystore`. The go-ethereum project allows safely encrypting keys with a passphrase before storing them on disk. The keystore is responsible for loading and decrypting the keys using the user-provided passphrase. To reduce the window of vulnerability, go-ethereum zeroes-out, in memory, decrypted keys after signing a hash or a transaction.

As a proof-of-concept, we implemented a simplified version of this keystore with GOTEE. The keystore executes in the enclave and enables: (1) loading an encrypted private key from the disk in the enclave, (2) decrypting the private key using a user-provided passphrase (e.g., via a secured ssh connection), and (3) signing a hash if the user validates it. Our keystore is 500 lines of Go code. The primary benefit of this approach is the elimination of the window of vulnerability. The keystore can safely keep private keys cached in secure memory. It took a single developer one day to implement this simplified secured keystore.

The enclave size break down is reported in Table 1. The amount of code loaded in the enclave, more than 1MB, is large compared to other experiments. This is mostly due the embedded ssh server, the cryptographic libraries, *e.g.,* elliptic curves and AES, and the encoding libraries, required to unmarshal decrypted private keys.

Along the TLS benchmark, this implementation validates that GOTEE can support popular Go cryptographic libraries (RSA, AES, and ECDSA) without modifying these packages.

6 Discussion

GOTEE demonstrates that language support for TEEs can alleviate SGX limitations and that the GOTEE programming model can be used to effectively increase the integrity and confidentiality of sensitive server-side computations. At the same time, the viability of SGX, beyond simple use cases in digital-rights management, as a foundational *trust* technology is doubtful given the large number of SGX vulnerabilities found to date and the complexity of the current architecture. SGX is a complex extension to a complex instruction set with an optimized implementation. Verifying the correctness of this extension and of its interactions with the large number of existing instructions is challenging [11, 26]. SGX has already been shown to be vulnerable to side-channel attacks based on caches [20], page faults [49], branch shadowing [38], and processor side-channel attacks ("Foreshadow" [22], a variant of Spectre [36] and Meltdown [41]).

GOTEE's increased isolation and decoupling between trusted and untrusted code, as well as the channel abstraction as the sole mean of communication, allows GOTEE applications to remain agnostic to the underlying technology's programming model. GOTEE seems ideally suited to provide a programming model for more radical TEE designs, that better protect trusted code in isolated environments comprising dedicated cores, TLBs, and (larger) dedicated, encrypted DRAM. One such TEE design could allocate processors and memory at kernel boot time. With a reserved co-processor, its TLB could be dedicated to an enclave and the responsibility of managing the virtual address space could shift from the operating system kernel to a kernel driver, with a small and verifiable implementation. A robust solution would also partition the cache hierarchy to avoid cache-based side-channel attacks.

7 Related Work

A GOTEE-compiled program results in side-by-side execution of two peer environments that communicate over type-

checked, message-passing channels. Using language-based message passing to isolate parts of a program is similar to the Singularity operating system [35], which used strongly typed channels as its only communication mechanism among processes and the kernel.

Program partitioning has been used to transform programs to run sensitive computations on isolated or secure processors. The Jif/split [50] system used security types and information-flow analysis to partition programs so that secure computations could be distributed and executed on trusted processors. Swift [25] partitioned a web app to run its trusted computation on a server. Wedge [19] was a Linux extension that supported least-privileged partitioning and execution of programs. The Crowbar tool used static program analysis to partition programs so that operations could be performed with least privilege. Privtrans [21] partitioned a program to enforce privilege separation. GOTEE, inspired by these systems, provides a language-base, compiler-driven code and data partitioning for TEEs that presents a simple programming model and which could be extended to support other TEE hardware, as well as secured co-processor or remote execution setups.

TrustScript [30] provides language support for running TypeScript (JavaScript) code in an enclave. Similar to GOTEE, it relies on keyword annotation of trusted code and uses asynchronous message passing between the trusted and untrusted runtimes. Unlike GOTEE's automated, fine-grain partitioning, TrustScript developers must implement all trusted code in an annotated namespace, and the TrustScript's security model is unclear.

Glamdring [40] uses data-driven code partitioning between an SGX enclave and an untrusted environment. The compiler and toolchain try to reduce the number of enclave crossings by bringing more code into the enclave. GOTEE takes a different approach, as it provides programmers with fine-grained control over the TCB, a stricter memory isolation between the two domains, and replaces enclave crossings with channel communications.

The debate on the relative merits of the crossing-oriented abstractions of the Intel SDK and the communication-oriented abstraction of GOTEE is of course a new twist on the duality of shared memory and message passing [37]. While numerous systems have been built with the domain-crossing approach embodied in the Intel SDK (§2.2) [8, 17, 40], the current implementation of SGX favors an asynchronous, communication-oriented model, as demonstrated by GOTEE and Intel's own recent `switchless` [14]. Other mentioned solutions [14, 16, 23, 30, 31, 44] rely *internally* on message passing to avoid enclave crossings. GOTEE, however, leverages Go channels, an abstraction that is part of a language, type-safe, and widely used. The cross-domain channels extend the general channel programming abstraction and enable developers to use *explicit* cross-domain communication at the application-level. Internally, this single point of interaction allows to perform both static and dynamic safety checks in concordance with the language semantics.

As a general result, GOTEE shows that programming language support, with an appropriate abstraction and programming model, combines the best of previous approaches, *i.e.,* the fine-grained automatic partitioning, the message passing model precluding enclave crossings, as well as a higher level of isolation between the two domains, and provides an interesting testbed for future extensions, such as information flow control or user-defined system call filtering.

Microsoft used SGX in conjunction with machine-code modification and verification to ensure a property called information release confinement that guarantees that attackers can only see encrypted data [48]. Although their C++ programming model is crossing oriented, GOTEE would provide a better starting point as they impose and verify safety restriction on the C++ enclave code that would be unnecessary for a safe language such as Go. Similarly, the Microsoft VC3 [46] map-reduce system requires and checks at run-time an even stronger set of control-flow and memory-safety properties, which again are easily satisfied by Go programs.

Finally, there exist software solutions which rely on layered virtualization to remove any trust dependency from the operating system [27, 28, 34] or the cloud hypervisor [52]. GOTEE could provide a complementary application-level isolation.

8 Conclusion

What comes first, the processor or the programming model? Intel's SGX made a TEE generally available, and its SDK provides a thin veneer that exposes its hardware features as the programming model. As systems are constructed on SGX, it has become increasingly clear that the most effective use of this TEE is to have it execute only trusted operations and to run the bulk of an application outside of the enclave. This paper explores a new programming model to support this style of use. GOTEE provides language support for TEEs. It extends the Go programming language and uses the Go routine mechanism to invoke a function within the enclave. Our compiler uses a single annotation to distinguish trusted code and automatically partition a program and establish cross-domain communication.

GOTEE treats the enclave as a distinct computing entity and uses message passing to copy arguments to functions, which then execute securely in a distinct, secure domain. This alternative model has the advantage of not requiring expensive cross-domain control transfers, resulting in significantly higher performance than the standard option. Equally important, it reduces the close coupling between the trusted and untrusted domains and opens the possibility of new, more easily verified hardware implementations that can better isolate TEE cores and run faster.

Acknowledgments

We thank Mathias Payer, Pascal Felber, the ATC anonymous reviewers, and our shepherd Christof Fetzer for their detailed comments. Moreover, we would like to thank Marios Kogias, George Prekas, and Jonas Fietz for the many discussions and constant feedback that lead to this paper. This work was funded in part by a VMware Research Grant.

References

[1] CVE-2016-5195 - write to read-only memory mappings. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195.

[2] CVE-2017-1000366 - glibc vulnerability leading to arbitrary code execution. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000366.

[3] CVE-2017-4948 - vmware out-of-bound read leads to confidentiality violation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-4948.

[4] CVE-2018-2727 - vulnerability in oracle financial services applications. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2727.

[5] CVE-2018-7160 - node.js dns rebind leads to full code execution access. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7160.

[6] Intel SGX - software guard extensions programming references. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[7] Intel SGX SDK - software guard extension, software development kit. https://software.intel.com/en-us/sgx-sdk.

[8] Rust sgx sdk. https://github.com/baidu/rust-sgx-sdk.

[9] Trustzone - arm. https://www.arm.com/products/security-on-arm/trustzone.

[10] The USA Patriot Act. https://www.justice.gov/archive/ll/highlights.htm, 2001.

[11] Intel Software Guard Extension (ISCA Tutorial). https://software.intel.com/sites/default/files/332680-002.pdf, 2015.

[12] CLOUD Act - H. R. 4943. https://www.congress.gov/bill/115th-congress/house-bill/4943/text, 2019.

[13] Ethereum project. https://www.ethereum.org/, 2019.

[14] Intel SGX Switchless - set of features to avoid expensive crossings. https://github.com/intel/linux-sgx/blob/master/sdk/switchless/, 2019.

[15] Adrien Ghosn, EPFL DCSL. GOTEE – a fork of Go with support for 'gosecure'. https://github.com/epfl-dcsl/gotee, 2019.

[16] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D. M., Kapitza, R., Pietzuch, P. R., and Fetzer, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 689–703.

[17] Baumann, A., Peinado, M., and Hunt, G. C. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[18] Berghel, H. Oh, What a Tangled Web: Russian Hacking, Fake News, and the 2016 US Presidential Election. *IEEE Computer 50*, 9 (2017), 87–91.

[19] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 309–322.

[20] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[21] Brumley, D., and Song, D. X. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 57–72.

[22] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 991–1008.

[23] che Tsai, C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., de Oliveira, D. A. S., and Porter, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 9:1–9:14.

[24] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 253–264.

[25] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web application via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 31–44.

[26] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[27] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. S. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)* (2014), pp. 81–96.

[28] DONG, X., SHEN, Z., CRISWELL, J., COX, A. L., AND DWARKADAS, S. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 1441–1458.

[29] ETHEREUM. Go Ethereum. https://github.com/ethereum/go-ethereum, 2019.

[30] GOLTZSCHE, D., SIEBELS, T., AND KAPITZA, R. Trustscript: Language support for partitioning trusted web applications. https://www.eurosys2019.org/wp-content/uploads/2019/03/eurosys19posters-abstract100.pdf, 2019.

[31] GOOGLE LLC. Asylo. https://asylo.dev/, 2019.

[32] GOOGLE LLC. gRPC. https://grpc.io/, 2019.

[33] HOARE, C. A. R. Communicating Sequential Processes. *Commun. ACM 21*, 8 (1978), 666–677.

[34] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 265–278.

[35] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *Operating Systems Review 41*, 2 (2007), 37–49.

[36] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *CoRR abs/1801.01203* (2018).

[37] LAUER, H. C., AND NEEDHAM, R. M. On the Duality of Operating System Structures. *Operating Systems Review 13*, 2 (1979), 3–19.

[38] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 557–574.

[39] LIANG, X., SHETTY, S., ZHANG, L., KAMHOUA, C. A., AND KWIAT, K. A. Man in the Cloud (MITC) Defender: SGX-Based User Credential Protection for Synchronization Applications in Cloud Computing Platform. In *Proceedings of the 10th IEEE International Conference on Cloud Computing (CLOUD)* (2017), pp. 302–309.

[40] LIND, J., PRIEBE, C., MUTHUKUMARAN, D., O'KEEFFE, D., AUBLIN, P.-L., KELBERT, F., REIHER, T., GOLTZSCHE, D., EYERS, D. M., KAPITZA, R., FETZER, C., AND PIETZUCH, P. R. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017), pp. 285–298.

[41] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *CoRR abs/1801.01207* (2018).

[42] MACKIE, K. Azure Confidential Computing Project Getting Added Partner Support. https://redmondmag.com/articles/2018/05/10/azure-confidential-computing-partners.aspx, 2018.

[43] MARLINSPIKE, M. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/.

[44] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 2017 EuroSys Conference* (2017), pp. 238–253.

[45] RUSSINOVICH, M. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017.

[46] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy* (2015), pp. 38–54.

[47] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security* (2007), pp. 552–561.

[48] SINHA, R., COSTA, M., LAL, A., LOPES, N. P., RAJAMANI, S. K., SESHIA, S. A., AND VASWANI, K. A design and verification methodology for secure isolated regions. In *Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation (PLDI)* (2016), pp. 665–681.

[49] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy* (2015), pp. 640–656.

[50] ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. C. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 1–14.

[51] ZEGZHDA, D. P., USOV, E. S., NIKOL'SKII, V. A., AND PAVLENKO, E. Use of Intel SGX to ensure the confidentiality of data of cloud users. *Automatic Control and Computer Sciences 51*, 8 (2017), 848–854.

[52] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. 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)* (2011), pp. 203–216.

[53] ZHAO, C., SAIFUDING, D., TIAN, H., ZHANG, Y., AND XING, C. On the Performance of Intel SGX. In *IEEE WISA* (2016), pp. 184–187.

Supporting Security Sensitive Tenants in a Bare-Metal Cloud*†

Amin Mosayyebzadeh[1], Apoorve Mohan[4], Sahil Tikale[1],
Mania Abdi[4], Nabil Schear[2], Charles Munson[2], Trammell Hudson[3]
Larry Rudolph[3], Gene Cooperman[4], Peter Desnoyers[4], Orran Krieger[1]
[1]Boston University [2]MIT Lincoln Laboratory [3]Two Sigma [4]Northeastern University

Abstract

Bolted is a new architecture for bare-metal clouds that enables tenants to control tradeoffs between security, price, and performance. Security-sensitive tenants can minimize their trust in the public cloud provider and achieve similar levels of security and control that they can obtain in their own private data centers. At the same time, Bolted neither imposes overhead on tenants that are security insensitive nor compromises the flexibility or operational efficiency of the provider. Our prototype exploits a novel provisioning system and specialized firmware to enable elasticity similar to virtualized clouds. Experimentally we quantify the cost of different levels of security for a variety of workloads and demonstrate the value of giving control to the tenant.

1 Introduction

There are a number of security concerns with today's clouds. First, virtualized clouds collocate multiple tenants on a single physical node, enabling malicious tenants to launch side-channel and covert channel attacks [21, 51, 54, 55, 66, 69, 80] or exploit vulnerabilities in the hypervisor to launch attacks both on tenants running on the same node [49, 64] and on the cloud provider itself [73]. Second, popular cloud management software like OpenStack can have a trusted computing base (TCB) with millions of lines of code and a massive attack surface [38]. Third, for operational efficiency, cloud providers tend to support *one-size-fits-all* solutions, where they apply uniform solutions (e.g. network encryption) to all customers; meeting the specialized requirements of highly security sensitive customers may impose unacceptable costs for others. Finally, and perhaps most concerning, existing clouds provide tenants with very limited visibility and control over internal operations and implementations; the tenant needs to fully trust the non-maliciousness and competence of the provider.

While bare-metal clouds [27, 46, 48, 63, 65] eliminate the security concerns implicit in virtualization, they do not address the rest of the challenges described above. For example, OpenStack's bare-metal service still has all of OpenStack in the TCB. As another example, existing bare-metal clouds ensure that previous tenants have not compromised firmware by adopting a one-size-fits-all approach of validation/attestation or re-flashing firmware. The tenant has no way to programmatically verify the firmware installed and needs to fully trust the provider. As yet another example, existing bare-metal clouds require the tenant to trust the provider to scrub any persistent state on the physical machine before allocating the machine to other tenants.[1]

These issues are a major concern for "security-sensitive" organizations, which we define as entities that are both willing to pay a significant price (dollars and/or performance) for security and that have the expertise, desire, or requirement to trust their own security arrangements over those of a cloud provider. Many medical, financial and federal institutions fit into this category. Recently, IARPA, who represents a number of such entities, released an RFI [12] that describes their requirement for using future public clouds; to "*replicate as closely as possible the properties of an air-gapped private enclave*" of physical machines. More concretely[2] this means a cloud where the tenant trusts the provider to make systems available but where confidentiality and integrity for a tenant's *enclave* is under the control of the tenant who is free to implement their own specialized security processes and procedures.

By our definition the majority of computing demands are not highly security-sensitive, thus providing a high-security option within a commercially-viable future cloud must not

*Mosayyebzadeh, Mohan, and Tikale contributed equally.

†DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

[1]See, for example, IBM Cloud's security policy for scrubbing local drives here https://tinyurl.com/y75sakn4. Note that scrubbing local disks can require hours of overhead on transferring computers between tenants; dramatically impacting the elasticity of the cloud.

[2]Per private communications with RFI authors.

impact the efficiency of providing service to other tenants. Is this possible? Can we make a cloud that is appropriate for even the most security sensitive tenants? Can we make a cloud where the tenant does not need to fully trust the provider? Can we do this without performance impact on tenants who are happy with the security levels of today's clouds?

The Bolted architecture and prototype implementation, described in this paper, demonstrates that the answer to these questions is "yes." The fundamental insight is that to implement a bare metal cloud only a minimum *isolation service* need to be controlled by the provider; all other functionality can be implemented by security-sensitive tenants on their own behalf, with provider-maintained implementations available to tenants with more typical security needs.

Bolted defines a set of micro-services, namely an *isolation service* that uses network isolation technologies to isolate tenants' bare-metal servers, a *provisioning service* that installs software on servers using network mounted storage, and an *attestation service* that compares measurements (hashes) of firmware/software on a server against a whitelist of allowed software. All services can be deployed by the provider as a one-size-fits-all solution for the tenants that are willing to trust the provider.

Security sensitive tenants can deploy their own provisioning and attestation service thereby minimizing their trust in the provider. The tenant's own software executing on machines (already trusted by the tenant), can validate measurements of code to be executed on some newly allocated server against her expectations rather than having to trust the provider. Further, the tenant's attestation service can securely distribute keys to the server for network and disk encryption. Using the default implementation of Bolted services, a tenant's enclave is protected from previous users of the same servers (using hardware-based attestation), from concurrent tenants of the cloud (using network isolation and encryption), and from future users of the same servers (using network mounted storage, storage encryption, and memory scrubbing). Further, a tenant with specialized needs can modify these services to match their requirements; the provider does not sacrifice operational efficiency or flexibility for security-sensitive customers with specialized needs since it is the tenant and not the provider responsible for implementing complex policies.

Key contributions of this paper are:

An **architecture** for a bare-metal cloud that: 1) enables security-sensitive tenants to control their own security while only trusting the provider for physical security and availability while 2) not imposing overhead on tenants that are security insensitive and not compromising the flexibility or operational efficiency of the provider. Key elements of the architecture are: 1) disk-less provisioning that eliminates the need to trust the provider for disk scrubbing (as well as the huge cost), 2) remote attestation (versus validation or re-flashing) to provide the tenant with a proof of the firmware and software running on their server and 3) secure deterministically built firmware that allows the tenant to inspect the source code used to generate the firmware.

A **prototype implementation** of the Bolted architecture where all its components are made available by us open-source, including the isolation service (Hardware Isolation Layer [5, 36]), a deterministic Linux-based minimal firmware (LinuxBoot [6,39], a disk-less bare-metal provisioning service (Bare Metal Imaging [7, 57]), a remote attestation service (Keylime [9, 72]), and scripts that interact with the various services to elastically create secure private enclaves. As we will discuss later, only the microservice providing isolation (i.e., Hardware Isolation Layer) is in the TCB and we show that this can, in fact, be quite small; just over 3K LOC in our implementation.

A **performance evaluation** of the Bolted prototype that demonstrates: 1) elasticity similar to today's virtualized cloud ($\sim$3 minutes to allocate and provision a physical server), 2) the cost of attestation has a modest impact $\sim$25% on the provisioning time, 3) there is value for customers that trust the provider in avoiding extra security (e.g.,$\sim$200% for some applications), while 4) security-sensitive customers can still run many non-IO intensive applications with negligible overhead and even I/O intensive BigData applications with a relatively modest (e.g., $\sim$30%) degradation.

2 Threat Model

We describe the threats to the victim, a tenant renting bare-metal servers from the cloud, and describe approaches taken by Bolted to safeguards against them. We consider external entities (hackers), malicious insiders in the cloud provider's organization and all other tenants of the server—both past and future—as potential adversaries to the victim. We assume that the goal of the adversary is to steal data, corrupt data, or deny services to the victim by gaining access to the victim's occupied servers or network. Our goal is to empower the tenant with the ability to take control of its own security; it is up to the tenant to make the tradeoff decision between the degree to which it relies on the provider's security systems versus the harm that it may suffer from a successful attack.

The cloud provider is always trusted with the physical security of the datacenter, thus any attacks involving physical access to the infrastructure, including power and noise analysis, bus snooping, or decapping chips [33, 34, 74] are out of scope of a tenant's control. The provider is also trusted for the availability of the network, node allocation services, and any network performance guarantees. We assume that the cloud itself is vulnerable to exploitation by external entities (hackers) or a malicious insider (e.g., a rogue systems administrator) but we trust the cloud provider's organization to have necessary systems and procedures in place to detect and limit the impact of such events. For example, the provider can enforce sufficient technical separation of duties (e.g., two-person rule) such that a single malicious insider or hacker cannot both re-flash all the node firmware in a data center and change what

hashes the provider publishes for attestation, have both physical and logical access to a node, or make unreviewed changes to the provider's deployed software, etc. Further, we assume that all servers in the cloud are equipped with a Trusted Platform Module (TPM) - a dedicated cryptographic coprocessor required for hardware-based authentication [11].

We categorize the threats that the tenant faces into the following phases:

Prior to occupancy: Malicious (or buggy) firmware can threaten the integrity of a server, as well as that of other servers it is able to contact. A tenant server's firmware may be infected prior to the tenant using it, either by the previous tenant (e.g., by exploiting firmware bugs) or by the cloud provider insider (e.g., by unauthorized firmware modification). If a server is not sufficiently isolated from potential attackers there is also a threat of infection between the time it is booted until it is fully provisioned and all defenses are in place.

During occupancy: Although many side-channel attacks are avoided by disallowing concurrent tenants on the same server, if the server's network traffic is not sufficiently isolated, the provider or other concurrent tenants of the cloud may be able to launch attacks against it or eavesdrop on its communication with other servers in the enclave. Moreover, if network attached storage is used (as in our implementation) all communication that is not sufficiently secured between server and storage may be vulnerable. Finally, there is a threat to the tenant from denial of service attacks.

After occupancy: Once the tenant releases a server, the confidentiality of a tenant may be compromised by any of its state (e.g, storage or memory) being visible to subsequent software running on the server.

3 Design Philosophy

The key goals of Bolted are: (1) to minimize the trust that a tenant needs to place in the provider, (2) to enable tenants with specialized security expertise to implement the functionality themselves, and (3) to enable tenants to make their own cost/performance/security tradeoffs – in bare-metal clouds. These goals have a number of implications in the design of Bolted.

First, Bolted differs from existing bare metal offerings in that most of the component services that make up Bolted can be operated by a tenant rather than by the provider. A security sensitive tenant can customize or replace these services. All the logic that orchestrates how different services are used to securely deploy a tenant's software is implemented using scripts that can be replaced or modified by the user. Most importantly, the service that checks the integrity of a rented server can be deployed (and potentially re-implemented) by the tenant.

Second, while we expect a provider to secure and isolate the network and storage of tenants, we only rely on the provider for availability and not for the confidentiality or integrity of the tenant's computation. For tenants that do not trust the provider, we assume that Bolted tenants will further encrypt all communication between the their servers and between those servers and storage. Bolted provides a (user-operated) service to securely distribute keys for this purpose.

Third, we rely on attestation (measuring all firmware and software and ensuring that it matches known good values) that can be implemented by the tenant rather than just validation (ensuring that software/firmware is signed by a trusted party). This is critical for firmware which may contain bugs [24, 35, 40, 41, 70, 77] that can disrupt tenant security. Attestation provides a time-of-use proof that the provider has kept the firmware up to date. More generally, the whole process of incorporating a server into an enclave can be attested to the tenant. In addition, the tenant can continuously attest when the server is operating, ensuring that any code loaded in any layer of software (OS, applications and etc., and irrespective of who signed them) can be dynamically checked against a tenant-controlled whitelist.

Fourth, we have a strong focus on keeping our software as small as possible and making it all available via open source. In some cases, we have written our own highly specialized functionality rather than relying on larger function rich general purpose code in order to achieve this goal. For functionality deployed by the provider, this is critical to enable it to be inspected by tenants to ensure that any requirements are met. For example, previous attacks have shown that firmware security features are difficult to implement bug-free – including firmware measurements being insufficient [26], hardware protections against malicious devices not being in place [59], and dynamic root of trust (DRTM) implementation flaws [79]. Further, our firmware is deterministically built, so that the tenant can not only inspect it for correct implementation but then easily check that this is the firmware that is actually executing on the machine assigned to the tenant. For tenant-deployed functionality, small open source implementations are valuable to enable user-specific customization.

Finally, servers are assumed to be stateless with all volumes accessed on-demand over the network. This removes confidentiality or denial of service attacks by the provider or subsequent tenants of server inspecting or deleting a tenants disk state. Bare-metal clouds that support stateful servers need to either give the tenant the guarantee that a node will never be preempted (problematic in a pay-for-use cloud model) or ensure that the provider scrubs the disks (trusting the provider and potentially requiring hours with modern disks). As we will see, stateless servers also dramatically improve the elasticity of the service.

4 Architecture

Bolted enables tenants to build a secure enclave of bare-metal servers where the integrity of each server is verified by the tenant before it is allowed to participate in the tenant's enclave. During the allocation process, a server transitions through the following states: **free**, or not allocated, **airlock**, where

the integrity of the server is checked, after which it is either **allocated** to a tenant's secure enclave if it passes the integrity check or **rejected** if it fails. In this section, we discuss the Bolted components; their operations; the process of server allocation, attestation, and the degrees of freedom in deploying Bolted components to support different security requirements and use cases.

4.1 Components

Bolted consists of four components which operate independently and (in the highest-security and lowest-trust configurations) are orchestrated by the tenant rather than the provider.

Isolation Service: The Isolation Service exposes interfaces to (de)allocate servers and networks to tenants, and isolate and/or group the servers by manipulating a provider's networking infrastructure (switches and/or routers). Using the exposed interfaces, the servers are moved to *free* or *rejected* state as well – ensuring the servers are not part of any tenant-owned network. These interfaces are also used to move the servers to the *airlock* state (to verify if they have been compromised) or the *allocated* state (where they are available for the tenant).

The Isolation Service uses network isolation techniques instead of encryption-based logical isolation in order to enforce guarantees of performance and to provide basic protection against traffic analysis attacks. Since the operations performed by these interfaces (on the networking infrastructure) are privileged, the isolation service needs to be deployed by the provider; if a tenant does not trust the provider, it can further encrypt network traffic between their servers.

Secure Firmware: Secure firmware is crucial towards improving tenant's trust of the public cloud servers; it should consist of following properties. First, it should be open-source, so that it benefits from large community support in improving its features and fixing any bugs and vulnerabilities. Second, it should be deterministically built so that a tenant can build the firmware from verified source code and independently validate the provider-installed firmware. Third, it must scrub server memory prior to launching a tenant OS – if the server was preempted from a previous tenant, it must guarantee that the previous tenants' code and data is not present in the memory. Finally, it must provide an execution environment for the attestation agent in the airlock state.

We note that it is challenging to replace computer firmware; even major providers are often forced to install huge binary blobs signed by the hardware manufacturer with no access to the source code. When firmware cannot be replaced, we use the installed firmware for the minimum amount of time in order to download our own secure firmware – and the servers' pre-installed firmware must support trusted boot [20].

While the overall Bolted architecture design supports the attestation and security of both system firmware (e.g., BIOS or UEFI) and peripheral firmware (e.g., GPU, network card, etc.), there are no standardized and implemented methods to attest those peripheral firmware to an external party. Early attempts at standardization are underway, and we expect Bolted can leverage them when they mature [67].

Provisioning Service: This service is broadly responsible for three things – (1) initial provisioning of the server with the software stack (i.e. secure firmware and attestation agent) responsible for its attestation during the *airlock* state, (2) provisioning of the server during the *allocated* state (i.e. the server was successfully verified that it was not compromised) with the intended software stack i.e. the operating system and the relevant software packages, and (3) saving and/or deleting the servers' persistent state when a server is released.

The Provisioning Service can be deployed either by the provider or by tenants themselves. The latter option is valuable for security-sensitive tenants who do not want to trust the provider with their operating system images or who want to use their own (e.g., legacy) provisioning systems. The provisioning service must provision the servers in a stateless manner so that the tenants do not have to rely on (and trust) the provider to remove any persistent state after the server is released.

Attestation Service: The Attestation Service consists of two parts: an attestation agent that executes on the server to be attested, and an attestation server that maintains a prepopulated database of known reliable hash measurements of allowed firmware/software (i.e., a whitelist). This service is used during the *airlock* and *allocated* states. The Attestation Service can be deployed either by the provider or by the tenant.

During the *airlock* state, the attestation agent (downloaded from the Provisioning Service during initial provisioning) is responsible for sending *quotes*[3] of the firmware and any other software involved during the boot sequence to the attestation server to be matched against the whitelist. Depending on the attestation result obtained from the attestation server, the state of the attested server is changed to allocated or rejected. In the case when the computer firmware cannot be replaced, the trusted boot sequence measurement (until the secure firmware is executed) must be supplied by the provider. Obtaining this measurement is a one-time operation for each server, and this whitelist can be published publicly by the provider.

In the *allocated* state, the attestation agent (installed on the tenants' OS) can continuously verify the software stack running against the whitelist present on the attestation server (also referred as *Continuous Attestation*). For continuous attestation to work, the software stack should be configured such that it saves new measurements to the cryptoprocessor upon observing any change/modification/access. The attestation agent periodically sends the new hash measurements of software and configuration registered in the cryptoprocessor to the attestation server; if attestation fails (i.e., when any malicious activity is observed), the attestation server alerts the

[3] Hash measurements obtained from and signed by a secure cryptoprocessor such as TPM.

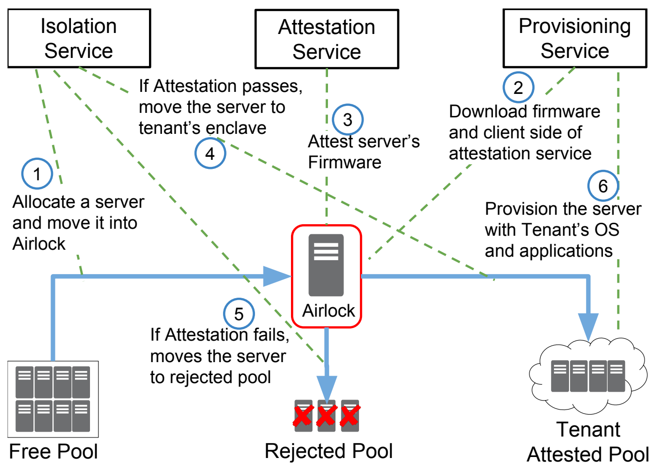


Figure 1: Bolted's Architecture: Blue arrows show state changes and green dotted lines shows the actions during a state change.

attestation agent. Continuous attestation protects tenants both against unauthorized execution of executables and against malicious reboots into unauthorized firmware, bootloader, or operating system. Note that continuous attestation is fundamentally more challenging in a provider-deployed attestation service, as the runtime whitelist (e.g., hashes of approved binaries allowed to be run on the node) must be tenant-generated; we assume continuous attestation is only used by security-sensitive tenants that deploy their own attestation service.

4.2 Life Cycle

The different Bolted components do not directly interact with each other, but instead, are orchestrated by user-controlled scripts. Figure 1 shows the **life-cycle of a typical secure server** (in the case of security-sensitive tenant), which progresses through six steps: (**1**) The tenant uses the Isolation Service to allocate a new bare metal server, create an airlock network, and move the server to that airlock, shared with the Attestation and the Provisioning networks; we need to isolate servers in the airlock state from other servers in the same state so that a compromised server cannot infect other uncompromised servers. (**2**) The secure firmware is executed (if stored in system flash) or provisioned onto the server along with a boot-loader, attestation software agent, and any other related software. With these in place, (**3**) the Attestation Service attests the integrity of the firmware of this server. Once initial attestation completes, (**4**) the tenant again employs the Isolation Service to move the server from the airlock network. If firmware attestation failed (**5**) it is moved into the Rejected Pool, isolated from the rest of the cloud; if attestation was successful, the server is made part of the tenant's enclave by connecting it to the tenant networks. In order to make use of the server, further provisioning is required (**6**) so the tenant again uses the Provisioning Service to install the tenant operating system and any other required applications.

4.3 Use Cases

Figure 2 demonstrates the flexibility of Bolted using three examples of users, namely; 1) Alice, a graduate student, who wants to maximize performance and minimize cost and does not care about security, 2) Bob, a professor, who does not trust other tenants (e.g., graduate students) but is willing to trust the provider, and 3) Charlie, a security-sensitive tenant, who not only does not trust other tenants but wants to minimize his trust in the provider.

Alice and Bob are willing to trust the provider's network isolation and storage security, and do not need to employ runtime encryption and will not incur its performance burden; nor will they need to expend the effort to deploy and manage their own services[4]. Alice, further, uses scripts that do not even bother using the provider's attestation service, further improving the speed that she can start up servers as well as her costs if the provider charges her for all the time a server is allocated to her.

Security-sensitive tenant Charlie deploys his own, potentially modified, provisioning and attestation service. He does not have to rely on the provider's network isolation to protect his confidentiality and integrity but can implement runtime protections such as network and disk encryption. Moreover, the attestation service can be used not only to protect him from previous tenants, but also to maintain a whitelist of applications and configuration, and to quickly detect any compromises in an ongoing fashion. The one area where Bolted requires Charlie to trust the provider is for protecting against denial of service attacks since only the provider can deploy the isolation service that allocates servers and controls provider switches. Trusting a provider, in this case, is unavoidable with current networking technology, as the provider controls all networking to the datacenter.

In addition to the cloud use cases, Bolted was designed to be flexible enough to handle the use case of co-location facilities [1, 4, 8] where the datacenter tenants temporarily "loan" computers to each other to handle fluctuations in demand; and this use case is, in fact, the primary one for which Bolted is going into production currently. In this case, a single party may be both provider and tenant. As an example, one party might have a high demand on their HPC cluster, while another party has spare capacity in their IaaS cloud; the isolation service from the second party (the provider) could be used to provision servers for loan to the first party, with attestation and provisioning services (including provisioning-associated storage) provided by the first party (the user).

Since the different Bolted services are independent, being orchestrated by tenant scripts, it is straightforward for a tenant to use capacity from multiple isolation services. The attestation of Bolted is important to enable supporting untrusted environments (e.g., research testbeds) alongside production services. For tenants that use the standard Bolted provisioning

[4] Or mismanage, a more significant risk for less security-literate users.

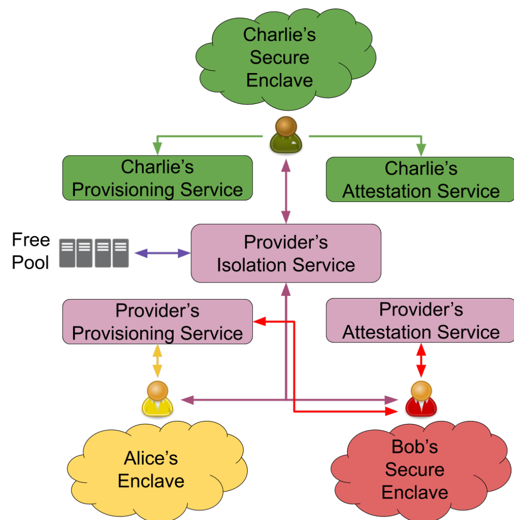


Figure 2: Bolted deployment examples; purple boxes are provider-deployed and greens are tenant-deployed. Alice and Bob trust the provider-deployed infrastructure, while security-sensitive Charlie deploys its own.

service, the use of network mounted storage by Bolted enables them to use their own storage for persistence, making storage encryption unnecessary. Because Bolted enables tenants to deploy their own provisioning service, some tenants can use custom provisioning services which install to local storage.[5] When using their own infrastructure, the tenant and provider are in the same organization. In this case, tenants trust the provider, and hence network encryption is unnecessary. Tenants are willing to make agreements with trusted partners from whom they will be using servers; trusting the partner's isolation service makes network encryption unnecessary for communication with servers obtained from it.

5 Implementation

We describe our implementation of the Isolation Service (HIL [36]), Firmware (LinuxBoot [6]), Attestation Service (Keylime [72]), and Provisioning Service (BMI [57]), and explain how they work together as Bolted. All of these constituent services of Bolted are open-source packages and can be modified by tenants or providers to meet their specific requirements.

Hardware Isolation Layer: The fundamental operations Hardware Isolation Layer (HIL) provides are (i) allocation of physical servers, (ii) allocation of networks, and (iii) connecting these servers and networks. A tenant can invoke HIL to allocate servers to an enclave, create a management network between the servers, and then connect this network to any provisioning tool (e.g., [15, 19, 57, 62]). It can also let tenants create networks for isolated communication between servers and/or attach those servers to public networks made available by the provider. HIL controls the network switches of the cloud provider and provides VLAN-based [45] network isolation mechanism. HIL also supports a simple API for Baseboard Management Controller (BMC) operations like power cycling servers and console access; ensuring that users cannot attack the BMC. HIL cannot be deployed by tenants and must be deployed by the provider and is the only component shared by tenants, that is not attested to. In our effort to minimize this TCB we have worked hard to keep HIL very simple (approximately 3000 LOC).

Because the provider is trusted for physical isolation and security, it also acts as the source of truth for information on servers in two ways. First, it maps each server's HIL identity to a TPM identity by exporting the TPM's public Endorsement Key (EK) through administrator-modifiable metadata per server, ensuring that the tenant is able to confirm that the tenant she received is indeed the one she reserved thus protecting the tenant from any server spoofing attack. Second, HIL exposes the provider-generated whitelist of TPM PCR measurements, i.e., ones that relate to the platform components like BIOS/UEFI firmware and firmware settings.

LinuxBoot: LinuxBoot is our firmware implementation and bootloader replacement. It is a minimal reproducible build of Linux that serves as an alternative to UEFI and Legacy BIOS. LinuxBoot retains the vendor PEI (Pre-EFI environment) code as well as the signed ACM (authenticated code modules) that Intel provides for establishing the TEE (trusted execution environment). LinuxBoot replaces the DXE (Driver Execution Environment) portion of UEFI with open source wrappers, the Linux Kernel, and a flexible initrd based runtime. Advantages over stock UEFI include: 1) LinuxBoot's open-source Linux devices drivers and filesystems have had significantly more scrutiny than the UEFI implementations, 2) its deterministic build enables easy remote attestation with a TPM; a tenant can independently confirm that the firmware on a server corresponds to source code that they compile themselves, 3) it can use any Linux-supported filesystem or device driver, execute Linux shell scripts to perform remote attestation over secure network protocols and mount encrypted drives, simplifying integration into services like Bolted, 4) it is significantly faster to POST than UEFI; taking 40 seconds on our servers, compared to about 4 minutes with UEFI.

We chose LinuxBoot over alternatives like Tianocore [10] – an open source implementation of UEFI because unlike Tianocore it does not depend on hardware drivers provided by motherboard vendors. In addition to the driver dependency Tianocore also needs support of Firmware Support Package (FSP) from processor vendors which are closed source binaries or independent softwares like coreboot [2, 3] to function as a complete bootable firmware. LinuxBoot does use FSP however Heads which is our flavor of LinuxBoot is able to establish root of trust prior to executing FSP thus ensuring that FSP blob is measured into TPM PCR's. This protects from attacks that involve replacing a measured FSP with a malicious FSP. Additionally, while LinuxBoot and Tianocore

[5] In this case, provisioning time is much larger and tenants are responsible for scrubbing the local disk.

both are open source projects, LinuxBoot is based on Linux, a much more mature and widely used system with battle tested code.

We have modified LinuxBoot such that it scrubs memory before a tenant can use a server; a tenant that attests that LinuxBoot is installed is guaranteed that subsequent tenants will not gain control until the memory has been scrubbed since the only way for the provider, or another tenant, to gain control (or reflash the firmware) is to power cycle the machine which will ensure that LinuxBoot is executed. Scripts integrated with LinuxBoot download the attestation service's client side agent, download and kexec a tenant's kernel (only if attestation has succeeded), and obtain a key from the attestation service to access the encrypted disk and network.

Keylime: Keylime is our remote attestation and key management system. It is divided into four major components: Registrar, Cloud Verifier, Agent, and Tenant. The registrar stores and certifies the public *Attestation Identity Keys (AIKs)* of the TPMs used by a tenant; it is only a trust root and does not store any tenant secrets. The Cloud Verifier (CV) maintains the whitelist of trusted code and checks server integrity. The Agent is downloaded and measured by the server (firmware or previously measured software) and then passes quotes (i.e., TPM-signed attestations of the integrity state of the machine) from the server's TPM to the verifier. The Tenant starts the attestation process and asks the Verifier to verify the server. The Registrar Verifier and Tenant can be hosted by the tenant outside of the cloud or could be hosted on a physical system in the cloud. Keylime delivers the tenant kernel, initrd and scripts to the server (after attestation success) using a secure connection between the Keylime CV and Keylime agent. The script is executed by the agent to 1) make sure the server is on the tenant's secure network and 2) kexec into tenant's kernel and boot the server.

For tenants that do not trust the provider, Keylime supports automatic configuration for Linux Unified Key Setup (LUKS) [13] for disk encryption and IPsec for network encryption using keys bootstrapped during attestation and bound to the TPM hardware root-of-trust. Also, Keylime integrates with the Linux Integrity Measurement Architecture (IMA) [71] to allow tenants to continuously attest that a server was not compromised after boot. IMA continuously maintains a hash chain rooted in the TPM of all programs, libraries, and critical configuration files that have been executed or read by the system. The CV checks the IMA hash chain regularly at runtime to detect deviations from the whitelist of acceptable hashes.

Bare Metal Imaging: The fundamental operations provided by the Bare Metal Imaging (BMI) are: (i) disk image creation, (ii) image clone and snapshot, (iii) image deletion, and (iv) server boot from a specified image. Similar to virtualized cloud services, BMI serves images from remote-mounted boot drives, with server access via an iSCSI (TGT [76]) service managed by BMI and back-end storage in a Ceph [78] distributed storage system. When the server network-boots, it only fetches the parts of the image it uses (less than 1% of the image is typically used), which significantly reduces the provisioning time [57]. BMI allows tenants to run scripts against a BMI-managed filesystem which we use to extract boot information (kernel, initramfs image and kernel command lines) from images so that they could be passed to a booting server in a secure way via Keylime.

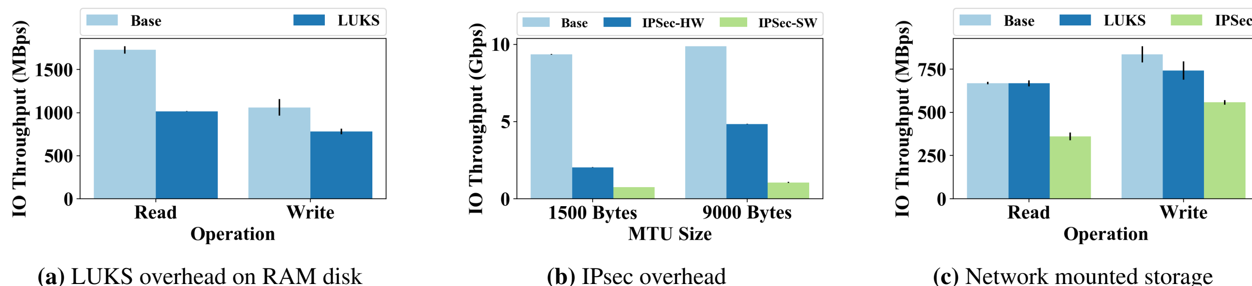
Putting it together: The booting of a server is controlled by a Python application that follows the sequence of steps shown in Figure 1. For servers that support it, we burn LinuxBoot directly into the server's SPI flash. Figure 1 shows another case where we download LinuxBoot's runtime (Heads) using iPXE and then continue the same sequence as if LinuxBoot was burned into the flash. We have modified the iPXE client code to measure the downloaded LinuxBoot runtime image into a TPM platform configuration register (PCR) so that all software involved in booting a server can be attested. When servers pass attestation, the Keylime Agent downloads an encrypted zip file containing the tenant's kernel, initrd, and a script from Keylime server and unzips them. The zip file also includes the keys for decrypting the storage and network. After a server is moved (using HIL) to the tenant's enclave, the Keylime Agent runs the script file. The script stores the cryptographic keys into an initrd file to pass it to the tenant's kernel and then kexecs into the downloaded kernel. After it boots, the kernel uses the keys from the initrd file to decrypt the remote disk and encrypt the network.

Keylime [72] and LinuxBoot [6] were previously created in part by authors of this paper, and modified as discussed above. While previously published, HIL [36] and BMI [57] were designed with the vision of integrating them in the larger Bolted architecture described in this paper.

6 Addressing the Threat Model

Here we discuss how, for security-sensitive tenants, Bolted's architecture addresses the threats in the three phases described in Section 2.

Prior to occupancy: We must protect a tenant's server against threats from previous users of the server and isolate it from potential network-based attacks until a server is fully provisioned. To do this, Bolted uses attestation to ensure that the firmware of the server was not modified by previous tenants, and isolates the server in the airlock state (protected from other tenants) until this attestation is complete. The deterministic nature of LinuxBoot enables tenants to inspect the source code of the firmware, and ensure that it is trusted, rather than just trusting the provider. Further, all communication within the networks in the enclave is encrypted, using a key provided by the tenant's attestation service (e.g., Keylime), ensuring that the server will not be susceptible to attacks by other servers as it is provisioned. Since our current implementation is unable to attest the state of peripheral firmware, there could be malware embedded in those devices that could compromise



(a) LUKS overhead on RAM disk

(b) IPsec overhead

(c) Network mounted storage

Figure 3: Performance Impact of Encryption

the node. Disk and network encryption securely bootstrapped by the TPM mitigate data confidentiality and integrity attacks from malicious peripherals with external access like network interfaces and storage controllers. System level isolation of device drivers, as in Qubes[6], could further be used to mitigate the impact of malicious peripherals mounting attacks against the node [32].

During occupancy: We must ensure that the server's network traffic is isolated so that the provider or other concurrent tenants of the cloud cannot launch attacks against it or eavesdrop on its communication with other servers in the enclave. HIL performs basic VLAN-based isolation to provide basic protection from traffic analysis by other tenants. However, a tenant can choose to both encrypt their network traffic with IPsec and shape their traffic to resist traffic analysis from the provider and not rely on provider's HIL. Keylime securely sends the keys for encrypting networking and disk traffic directly to the node. Disk encryption ensures the confidentiality and integrity of the persistent data even if the storage is under the control of a malicious provider.

Continuous attestation can detect changes to the runtime state of the server (e.g., unauthorized binaries being executed or reboot to an unauthorized kernel) and notify the tenant to take some action to respond. Response actions include revoking the cryptographic keys used by that server for network/storage encryption, removing it from the enclave VLAN, and immediately rebooting the system into a known good state and scrubbing its memory. While IMA only supports load/read-time measurement (i.e., hashing) of files on the system as they are used, most existing runtime protection measures like kernel integrity monitoring [56], control-flow integrity [25], or dynamic memory layout randomization [23] are built into either the kernel image/modules, application binaries, or libraries themselves. Thus, TPM measurements created by IMA at runtime will demonstrate that those protections were loaded.

After occupancy: Once a server is removed from a tenant enclave, we must ensure that the confidentiality of a tenant is not compromised by any of its state being visible to subsequent software running on the server. Stateless provisioning of the servers protects against any persistent state remaining on the server and avoids any reliance on the provider scrubbing the disk if it preempts the tenant. Further, the tenant can deploy its own provisioning service and ensure that the provider has no access to that storage. If the tenant requires the use of the local disks for performance reasons (e.g., for big data applications), the server can use local disk encryption with ephemeral keys stored only in memory. As long as the tenant attests that LinuxBoot is used, it knows that this firmware will zero the server's memory before another tenant will have the opportunity to execute any code.[7]

[6]https://www.qubes-os.org/

[7]Note that we are assuming here that the provider cannot re-flash the firmware remotely over the BMC.

7 Evaluation

We first use micro-benchmarks to quantify the cost of encrypted storage and networking on our system, then examine the performance and scalability of the Bolted prototype, the cost of continuous attestation and finally the performance of applications deployed using Bolted under different assumptions of trust.

7.1 Infrastructure and methodology

Single server provisioning experiments were performed on a Dell R630 server with 256 GB RAM and 2 Xeon E5-2660 v3 2.6GHz processors with 10 (20 HT) cores, using UEFI or LinuxBoot executing from motherboard flash. All the other experiments were conducted on a cluster of 16 Dell M620 blade servers (64 GB memory, 2 Xeon E5-2650 v2 2.60GHz processors with 8 cores (16 HT) per socket) and a 10Gbit switch. The M620 servers do not have a hardware TPM, so for functionality, we used a software emulation of a TPM [43], and for performance evaluation, emulated the latency to access the TPM based on numbers collected from our R630 system.

HIL, BMI, and Keylime servers were run on virtual machines with Xeon E5-2650 2.60GHz CPUs: Keylime with 16 vCPUs and 8GB memory; BMI with 2 vCPUs and 8GB, and HIL with 8 vCPUs and 8GB RAM. The iSCSI server ran on a virtual machine with 8 vCPUs and 32GB RAM. The Ceph cluster (the storage backend for BMI disk images) has 3 OSD servers (each dual Xeon E5-2603 v4 1.70GHz CPUs, 6 cores each) and a total of 27 disk spindles across the 3 machines. The servers were provisioned with Fedora 28 images (Linux

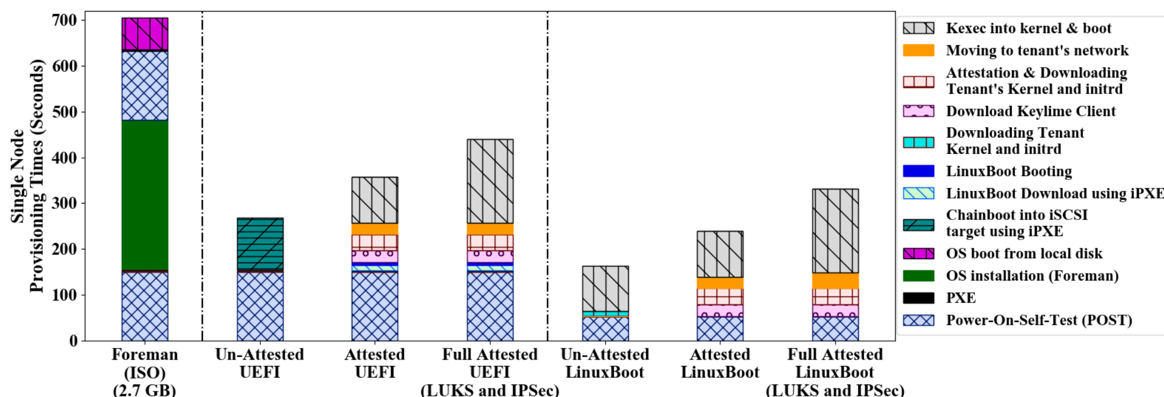


Figure 4: Provisioning time of one server.

kernel 4.17.9-200) enabled with IMA and version 5.6.3 of Strongswan [17] for IPsec. IPsec was configured in 'Host to Host' and Tunnel mode. The cryptographic algorithm used was AES-256-GCM SHA2-256 MODP2048. The authentication and encryption were done through a pre-shared key (PSK). IMA used SHA-256 hash algorithm. Cryptsetup utility version 1.7.0 was used to setup disk encryption based on LUKS – with AES-256-XTS algorithm. Unless otherwise stated, each experiment was executed five times.

7.2 The cost of encryption

For security-sensitive tenants that do not trust a provider, they must encrypt the disk and network traffic. To understand the overhead in our environment, we ran some simple micro-benchmarks.

Disk Encryption: Figure 3a shows the overhead of LUKS disk encryption on a Block RAM disk exercised using Linux's "dd" command. While LUKS introduces overhead in this extreme case, we can see that the bandwidth that LUKS can sustain at 1GB for reads is likely to be able to keep up with both local disks and network mounted storage delivered over a 10Gbit network while write performance may introduce a modest degradation at ∼0.8GB.

Network Encryption: Figure 3b shows the overhead of IPsec using Iperf between two servers using both hardware-based Intel AES-NI (IPsec HW) and software-based AES (IPsec SW) and MTU's of 1500 and 9000. We can see that IPsec has a much larger performance overhead than LUKS disk encryption, with even the best case of HW accelerated encryption and jumbo frames having almost a factor of two degradation over the non-encrypted case (CPU usage on our infrastructure is between 60% and 80% of one processing core for HW accelerated encryption). Additional tuning or specialized IPsec acceleration network interfaces could be used to boost performance [37]. We use hardware accelerated encryption and jumbo frames for all subsequent experiments.

Network mounted storage: In our implementation we boot servers using iSCSI which in turn accesses data from our Ceph cluster. In Figure 3c we show the results of exercising the iSCSI server using "dd". Experimentally, we found that increasing the read ahead buffer size on Linux to 8MB was critical for performance, and we do this on all subsequent experiments (the default size is 128KB). Since Ceph as the backend storage reads data in 4MB chunks, increasing the read ahead buffer size to 8MB results in higher sequential read performance. As expected we find that LUKS introduces small overhead on writes and no overhead on reads, while IPsec between the client and iSCSI server has a major impact on performance.

7.3 Elasticity

Today's bare-metal clouds take many tens of minutes to allocate and provision a server [61]. Further, scrubbing the disk can take many hours; an operation required for stateful bare metal clouds whenever a server is being transferred between one tenant and another. In contrast, virtualized clouds are highly elastic; provisioning a new VM can take just a few minutes and deleting a VM is nearly instantaneous. The huge difference in elasticity between bare-metal clouds and virtualized clouds has a major impact on the use cases for which bare-metal clouds are appropriate. How close can we approach the elasticity of today's virtualized clouds? What extra cost does attestation impose on that elasticity? What is the extra cost if the tenant does not trust the provider and need to encrypt disks and storage?

To understand the elasticity Bolted supports, we first examine its performance for provisioning servers under different assumptions of security and then examine the concurrency for provisioning multiple servers in parallel.

Provisioning time: Figure 4 compares the time to provision a server with Foreman (a popular provisioning system) [30] to Bolted with both UEFI and LinuxBoot firmware under 3 scenarios: *no attestation* which would be used by clients that are insensitive to security, *attestation* where the tenant trusts the provider, but uses (provider deployed) attestation to ensure that previous tenants have not compromised the server, and *Full attestation*, where a security-sensitive tenant that does not trust the provider uses LUKS to encrypt

the disk and IPsec to encrypt the path between the client and iSCSI server. There are a number of important high-level results from this figure. First for tenants that trust the provider, Bolted using LinuxBoot burned in the ROM is able to provision a server in under 3 minutes in the unattested case and under 4 minutes in the attested case; numbers that are very competitive with virtualized clouds. Second, attestation adds only a modest cost to provisioning a server and is likely a reasonable step for all systems. Third, even for tenants that do not trust the provider, (i.e. LUKS & IPsec) on servers with UEFI, Bolted at ∼7 minutes is still 1.6x faster than Foreman provisioning; note that Foreman implements no security procedures and is likely faster than existing cloud provisioning systems that use techniques like re-flashing firmware to protect tenants from firmware attacks.

Examining the detailed time breakdowns in Figure 4; while we introduced LinuxBoot to improve security, we can see that the improved POST time (3x faster than UEFI) on these servers has a major impact on performance. We also see that booting from network mounted storage, introduced to avoid trusting the provider to scrub the disk, also has a huge impact on provisioning time. The time to install data on to the local disk is much larger for the Foreman case, where all data needs to be copied into the local disk. In contrast, with network booting, only a tiny fraction of the boot disk is ever accessed. We also see that with a stateful provisioning system like Foreman, it needs to reboot the server after installing the tenant's OS and applications on the local disk of the server; incurring POST time twice. While not explicitly shown here, it is also important to note that with Bolted a tenant can shutdown the OS and release a node to another tenant and then later restart the image on any compatible node; a key property of elasticity in virtualized clouds that is not possible with stateful provisioning systems like Foreman.

We show in Figure 4 the costs of all the different phases of an attested boot. With UEFI, after POST, the phases are: (i) PXE downloading iPXE, (ii) iPXE downloading LinuxBoot's runtime (Heads), (iii) booting LinuxBoot, (iv) downloading the Keylime Agent (using HTTP), (v) running the Keylime Agent, registering the server and attesting it, and then downloading the tenant's kernel and initrd, (vi) moving the server into the tenant's network and making sure it is connected to the BMI server and finally (vii) LinuxBoot kexec'ing into the tenant's kernel and booting the server. In each step, the running software measures the next software and extends the result into a TPM PCR. Using LinuxBoot firmware, after POST we immediately jump to step (iv) above.

While the steps for attestation where complex to implement, the overall performance cost is relatively modest, adding only around 25% to the cost of provisioning a server.[8] This is an

[8]Moreover, given that performance is sufficient, we have so far made no effort to optimize the implementation. Obvious opportunities include better download protocols than HTTP, porting the Keylime Agent from python to Rust, etc.

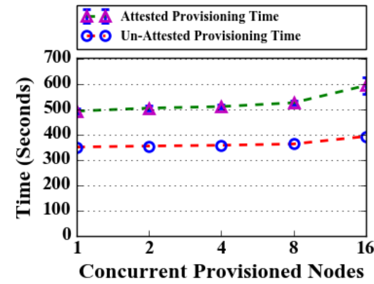


Figure 5: Bolted Concurrency

important result given a large number of bare-metal systems (e.g. CloudLab, Chameleon, Foreman, ...), that take no security measure today to ensure that firmware has not been corrupted. There is no performance justification today for not using attestation, and our project has demonstrated that it is possible to measure all components needed to boot a server securely. For the full attestation scenarios (UEFI and LinuxBoot), two more steps are added to the basic attestation scenarios: (+i) loading the cryptographic key and decrypting the encrypted storage with LUKS (+ii) establishing IPsec tunnel and connecting to the encrypted network. These two steps are incorporated into Kernel boot time in Figure 4. We can see that the major cost is not these extra steps but the slow down in booting into the image that comes from the slower disk that is accessed over IPsec.

Concurrency: Figure 5 shows (with UEFI firmware) how Bolted performs, with and without attestation, as we increase the number of concurrently booting servers (log scale). In both the attested and unattested case performance stays relatively flat until 8 nodes. In our current environment, this level of concurrency/elasticity has been more than sufficient for the community of researchers using Bolted. There is a substantial degradation in both the attested and unattested case when we go from 8 to 16 servers. In the unattested case, the degradation is due to the small scale Ceph deployment (with only 27 disks) available in our experimental infrastructure. For the attested boot, the performance degradation arises from a limitation in our current implementation where we only support a single airlock at a time; attestation for provisioning is currently serialized. While this scalability limitation is not a problem for current use cases in our data center, we intend to address it to enable future use cases of highly-elastic security-sensitive tenants; e.g., a national emergency requiring many computers.

7.4 Continuous Attestation

Once a server has been provisioned, a security sensitive tenant can further use IMA to continuously measure any changes to the configuration and applications. The Keylime Agent will include the IMA measurement list along with periodic continuous attestation quotes. This allows the Keylime Cloud Verifier to help ensure the integrity of the server's runtime state by comparing the provided measurement list with a

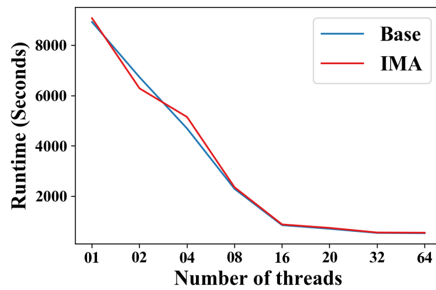


Figure 6: IMA overhead on Linux Kernel Compile

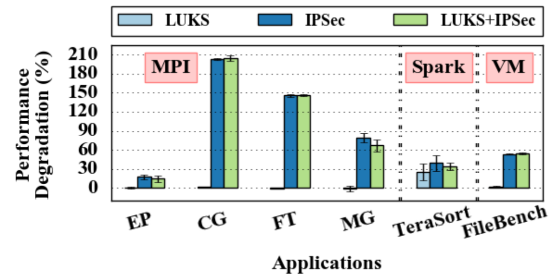


Figure 7: Macro-benchmarks' performance

whitelist of approved values provided by the tenant. In the case of a policy violation, Keylime can then revoke any keys used for network or disk encryption; essentially isolating the server. To evaluate IMA performance, we measured Linux kernel 4.16.12 compile time with and without IMA with a different number of processing threads. We use kernel compilation as a test case for IMA because it requires extensive file I/O and execution of many binaries. The IMA policy we used measured all files that are executed as well as all files read by the *root* user. To stress IMA we ran the kernel compile as root such that all of its activity would be measured.[9] Figure 6 shows the results in log scale; even in this unrealistic stress test IMA does not impose a noticeable overhead.

Keylime can detect policy violations from checking the IMA measurements and TPM quotes in under one second. To simulate a policy violation, we ran a script on the server without having a record of it in the whitelist, resulting in an IMA measurement different than expected. This results in Keylime issuing a revocation notification for the key of the affected server used for IPsec to the other servers in the system; the entire process takes approximately 3 seconds for a compromised server to have its IPsec connections to other servers reset and be cryptographically banned from the network.

7.5 Macro-Benchmarks

Security-sensitive tenants using Bolted rely on network and disk encryption to minimize their trust in the provider. Surprisingly there is little information in the literature what the cost of such encryption is for real applications. Is the performance good enough that we can tolerate a one-size-fits-all solution and avoid ever trusting the provider? Is the performance so poor that it will never make sense for security-sensitive customers to use Bolted?

Figure 7 (MPI) shows performance degradation results for a variety of applications from the NAS Parallel Benchmark [22] version 3.3.1: Embarrassingly Parallel (EP), Conjugate Gradient (CG), Fourier Transform (FT) and Multi Grid (MG) applications class D running in a 16 server enclave. We see overall that these applications only suffer significant overhead for IPsec, ranging from ∼18% for EB, which has modest communication, to ∼200% for CG which is very communication intensive. These results suggest that there are definitely workloads for which not trusting the provider incurs little overhead. At the same time, a one-size-fits-all solution is inappropriate; only tenants that are willing to trust the provider, and avoid the cost of encryption, are likely to run highly communication intensive applications in the cloud.

To understand the performance overhead for more cloud relevant workloads, Figure 7 (Spark) shows the performance of Spark [81] framework version 2.3.1 (working on Hadoop version 2.7.7) running TeraSort on a 260GB data set. The experiment is run in parallel in an enclave of 16 servers. TeraSort is a complex application which reads data from remote storage, shuffles temporary data between servers and writes final results to remote storage. We can see a significant overall degradation, of ∼30% for LUKS+IPsec. While this degradation is significant, we expect that security sensitive tenants would be willing to incur this level of overhead. On the other hand, this overhead is large enough that tenants willing to trust the provider would prefer not to incur it, suggesting that the flexibility of Bolted to provide this choice to the tenant is important.

Our last experiment (Figure 7 (VM)) is based on virtualization. An important application of bare metal servers is to run virtualized software (e.g., an IaaS cloud). In this experiment, we installed KVM QEMU version 2.11.2 on a M620 server as the hypervisor. The virtual machine we run on the hypervisor is CentOS 7 with Linux kernel 3.10.0. It has 8 vCPU cores and 32 GB RAM. This is based on the observation [28] that 90% of virtual machines having $\leq$ 8 vCPU cores and $\leq$ 32 GB RAM. We run Filebench version 1.4.9.1 benchmark [75] on 1000 files with 12MB average size on the virtual machine. We can see that the performance of this benchmark is ∼50% worse in the case of IPsec; a significant performance penalty. While we would expect less of a degradation for regular VMs (rather than ones running a file system benchmark), we can see that a tenant deploying generic services, like virtualization, should be very careful about the kind of workload they expect to use the service.

8 Related Work

Our work on creating a secure bare-metal cloud was motivated by a huge body of research demonstrating vulnerabilities due

[9] This policy and workload are very unlikely to be either useful or manageable from a security perspective. We used them only as a stress test.

to co-location in virtualized clouds including both hypervisor attacks [21,49,64,73,80] and side-channel and cover-channel attacks like the Meltdown and Spectre exploits [51,54,55,66, 69].

There is a large body of products and research projects for bare-metal clouds [27,46,48,63,65] and cluster deployment systems [15, 42, 62, 68] that have many of the capabilities of isolation and provisioning that Bolted includes. The fundamental difference with Bolted is that we strongly separate isolation from provisioning and different entities (e.g. security sensitive tenants) can control/deploy and even re-implement the provisioning service. This structuring clearly defines the TCB that needs to be deployed by the provider.

While it is often unclear exactly which technique each cloud uses to protect against firmware attacks, a wide variety of techniques have been used including specialized hardware [16, 18], using a specialized hypervisor to prevent access to firmware [31], and attestation to the provider [44, 47]. In all cases, there is no way for a tenant to programmatically verify that the firmware is up to date and not compromised by previous tenants. Bolted is unique in enabling tenant deployed attestation for bare-metal servers, where the measurement of the firmware and software are provided directly to the tenant.

The static root of trust (SRTM) approach used by Bolted requires all software to be measured in an unbroken chain of trust. It would have been simpler for us to use dynamic root of trust (DRTM), however, DRTM has additional chip dependencies and, more importantly, been shown to be vulnerable to attacks [79] and work of Kovah et. al has shown that it can be used as an attack vector itself [52].

9 Concluding Remarks

We presented Bolted, an architecture for a bare metal cloud that is appropriate for even the most security sensitive tenants; allowing these customers to take control over their own security. The only trust these tenants need to place in the provider is for the availability of the resources and that the physical hardware has not been compromised. At the same time, by delegating security for security sensitive tenants to the tenants, Bolted frees the provider from the complexity of having to directly support these demanding customers and avoids impact to customers that are less security sensitive.

To enable a wide community to inspect the TCB, all components of Bolted are open source. We designed HIL, for example, to be a simple micro-service rather than a general purpose tool like IRONIC [62] or Emulab [19]. HIL is being incorporated into a variety of different use cases by adding tools and services on and around it rather than turning it into a general purpose tool. Another key example of a small open source component is LinuxBoot. LinuxBoot is much simpler than UEFI. Since it is based on Linux, it has a code base that is under constant examination by a huge community of developers. LinuxBoot is reproducibly built, so a tenant can examine the software to ensure that it meets their security requirements and then ensure that the firmware deployed on machines is the version that they require.

Bolted protects against compromise of firmware executable by the system CPU; however modern systems may have other processors with persistent firmware inaccessible to the main CPU; compromise of this firmware is not addressed by this approach. These include: Base Management Controllers (BMCs) [58], the Intel Management Engine [29,53,60], PCIe devices with persistent flash-based firmware, like some GPUs and NICs, and storage devices [50]. Additional work (e.g. IOMMU based techniques, disabling the Management Engine [14] and the use of specialized systems with minimum firmware) will be needed to meet these threats.

The evaluation of our prototype has demonstrated that we can rapidly provision secure servers with competitive performance to today's virtualized clouds; removing one of the major barriers to bare metal clouds. We demonstrate that the cost of not trusting the provider (network/storage encryption) and of additional runtime security (continuous attestation) varies enormously depending on the application. (In fact, we are not aware of other work that has quantified the cost of network encryption, disk encryption, and continuous attestation with modern servers and implementation.) Results for HPC applications vary from negligible overhead to three times overhead for communication-intensive applications. Clearly the public cloud becomes economically unattractive for applications with three times overhead unless there are no other alternatives. However, we expect that the $\sim$30% degradation we see for TeraSort is likely representative of many applications today. Such overheads suggest that the cost of security is modest enough that security-sensitive customers will find value in using cloud resources. At the same time, the overhead is significant enough that the flexibility of Bolted that enables tenants to just pay for the security they need is justified. One surprising result is that our secure firmware, LinuxBoot achieves dramatically better POST time than existing firmware; this is one of the few times in our experience that additional security comes with performance advantages.

10 Acknowledgment

We would like to acknowledge the feedback of the anonymous reviewers and our shepherd, Dr. Nadav Amit. We would like to thank Red Hat, Two Sigma and NetApp, the core industry partners of Mass Open Cloud (MOC) for supporting this work. This project involved extensive efforts over many years to integrate all the components together. We gratefully acknowledge Jason Hennessey, Gerardo Ravago, Ali Raza, Naved Ansari, Kyle Hogan, and Radoslav Nikiforov Milanov for their significant contributions in development and their assistance in the evaluations. Partial support for this work was provided by the USAF Cloud Analysis Model Prototype project, National Science Foundation awards CNS-1414119, ACI-1440788 and OAC-1740218.

References

[1] ABOUT THE MGHPCC | MGHPCC. http://www.mghpcc.org/about/about-the-mghpcc/.

[2] coreboot - payloads. https://doc.coreboot.org/payloads.html.

[3] Coreboot minimal firmware. https://doc.coreboot.org/.

[4] Equinix Private Cloud Architecture. https://www.equinix.com/solutions/cloud-infrastructure/private-cloud/architecture/.

[5] Hil: Hardware Isolation Layer, formerly Hardware as a Service. https://github.com/CCI-MOC/hil.

[6] LinuxBoot. https://trmm.net/LinuxBoot_34c3.

[7] Malleable Metal as a Service (M2). https://github.com/CCI-MOC/M2.

[8] NWRDC | The Ultimate Solution to Simplify Your Data Center. http://www.nwrdc.fsu.edu/.

[9] python-keylime: Bootstrapping and Maintaining Trust in the Cloud. https://github.com/mit-ll/python-keylime.

[10] What is TianoCore? https://www.tianocore.org/.

[11] Trusted Platform Module (TPM) Summary. https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/, Apr. 2008.

[12] Creating a Classified Processing Enclave in the Public Cloud |IARPA. https://www.iarpa.gov/index.php/working-with-iarpa/requests-for-information/creating-a-classified-processing-enclave-in-the-public-cloud, 2017.

[13] Linux unified key setup. https://gitlab.com/cryptsetup/cryptsetup/blob/master/README.md, 2018.

[14] me_cleaner: Tool for partial deblobbing of intel me/txe firmware images. https://github.com/corna/me_cleaner, 2018.

[15] Metal as a service(maas) from canonical. https://maas.io/, 2018.

[16] Project Cerberus Architecture Overview. https://github.com/opencomputeproject/Project_Olympus/tree/master/Project_Cerberus, Dec 2018.

[17] Strongswan. https://www.strongswan.org/, Oct. 2018.

[18] Titan in depth: Security in plaintext. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext/, 2019.

[19] D. S. Anderson, M. Hibler, L. Stoller, T. Stack, and J. Lepreau. Automatic online validation of network configuration in the emulab network testbed. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 134–142. IEEE, 2006.

[20] W. A. Arbaugh. Trusted computing. *Department of Computer Science, University of Maryland,[online][Retrieved on Feb. 22, 2007] Retrieved from the Internet*, 2007.

[21] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. L. Porta, P. McDaniel, and L. Marvel. Malicious co-residency on the cloud: Attacks and defense. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[22] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[23] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 268–279, New York, NY, USA, 2015. ACM.

[24] Y. Bulygin, J. Loucaides, A. Furtak, O. Bazhaniuk, and A. Matrosov. Summary of attacks against BIOS and secure boot. *Defcon-22*, 2014.

[25] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, Apr. 2017.

[26] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. BIOS Chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 25–36, New York, NY, USA, 2013. ACM.

[27] I. Cloud. Bare metal servers. https://www.ibm.com/cloud/bare-metal-servers, 2018.

[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 Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, New York, NY, USA, 2017. ACM.

[29] M. Ermolov and M. Goryachy. How to hack a turned - off computer, or running unsigned code in intel management engine. https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf, Dec 2017.

[30] Foreman. Foreman. https://www.theforeman.org/, 2019.

[31] T. Fukai, S. Takekoshi, K. Azuma, T. Shinagawa, and K. Kato. BMCArmor: A Hardware Protection Scheme for Bare-Metal Clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 322–330, Dec 2017.

[32] P. Z. Gal Beniamini. Over the air: Exploiting Broadcom's wi-fi stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[33] M. Guri, B. Zadov, D. Bykhovsky, and Y. Elovici. PowerHammer: Exfiltrating Data from Air-Gapped Computers through Power Lines. *arXiv:1804.04014 [cs]*, Apr. 2018. arXiv: 1804.04014.

[34] 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, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60, 2008.

[35] J. Heasman. Rootkit threats. *Network Security*, 2006(1):18–19, 2006.

[36] J. Hennessey, S. Tikale, A. Turk, E. U. Kaynar, C. Hill, P. Desnoyers, and O. Krieger. HIL: Designing an exokernel for the data center. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, Santa Clara, CA, Oct. 2016.

[37] A. Hoban. Using intel® aes new instructions and pclmulqdq to significantly improve ipsec performance on linux. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/aes-ipsec-performance-linux-paper.pdf, August 2010.

[38] K. Hogan, H. Maleki, R. Rahaeimehr, R. Canetti, M. van Dijk, J. Hennessey, M. Varia, and H. Zhang. On the universally composable security of openstack. *IACR Cryptology ePrint Archive*, 2018:602, 2018.

[39] T. Hudson. Linuxboot. https://github.com/osresearch/linuxboot.

[40] T. Hudson, X. Kovah, and C. Kallenberg. ThunderStrike 2: Sith Strike. *Black Hat USA Briefings*, 2015.

[41] T. Hudson and L. Rudolph. Thunderstrike: EFI firmware bootkits for Apple Macbooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 15. ACM, 2015.

[42] IBM. Extreme Cloud Administration Toolkit — xCAT 2.14.5 documentation. https://xcat-docs.readthedocs.io/en/stable/index.html#, 2019.

[43] IBM. Ibm's tpm 1.2. http://ibmswtpm.sourceforge.net/, 2019.

[44] IBMcloud. Hardware monitoring and security controls. https://console.bluemix.net/docs/bare-metal/intel-trusted-execution-technology-txt.html#hardware-monitoring-and-security-controls, Apr 2018.

[45] IEEE Computer Society. *IEEE standard for local and metropolitan area networks media access control (MAC) bridges and virtual bridged local area networks*. Institute of Electrical and Electronics Engineers, New York, 2018.

[46] A. W. S. Inc. Amazon EC2 Bare Metal Instances with Direct Access to Hardware. https://aws.amazon.com/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/, 2017.

[47] O. Inc. Oracle Cloud Infrastructure Security. *Oracle Cloud Infrastructure white papers*, page 36, Nov 2018.

[48] Internap. Bare-metal AgileSERVER. http://www.internap.com/bare-metal/, 2015.

[49] S. T. King and P. M. Chen. Subvirt: Implementing malware with virtual machines. In *Security and Privacy, 2006 IEEE Symposium on*, pages 14–pp. IEEE, 2006.

[50] J. Kirk. Destroying your hard drive is the only way to stop this super-advanced malware. https://www.pcworld.com/article/2884952/equation-cyberspies-use-unrivaled-nsastyle-techniques-to-hit-iran-russia.html, Feb 2015.

[51] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[52] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell. SENTER Sandman: Using Intel TXT to Attack BIOSes. In *HITB Security Conference*, page 5, Amsterdam, May 2014.

[53] A. Kroizer. Tpm and intel ® ptt overview. http://tce.webee.eedev.technion.ac.il/wp-content/uploads/sites/8/2016/01/AK_TPM-overview-technion.pdf, Sep 2015.

[54] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[55] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[56] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07, pages 21–29, New York, NY, USA, 2007. ACM.

[57] A. Mohan, A. Turk, R. Gudimetla, S. Tikale, J. Hennessey, U. Kaynar, G.Cooperman, P. Desnoyers, and O. Krieger. M2: Malleable Metal as a Service. *ArXiv e-prints*, 2018.

[58] H. Moore. A penetration tester's guide to ipmi and bmcs. https://blog.rapid7.com/2013/07/02/a-penetration-testers-guide-to-ipmi/, Aug 2017.

[59] B. Morgan, E. Alata, V. Nicomette, and M. Kaâniche. Bypassing IOMMU protection against I/O attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150, Oct 2016.

[60] L. H. Newman. Intel chip flaws leave millions of devices exposed. https://www.wired.com/story/intel-management-engine-vulnerabilities-pcs-servers-iot/, Nov 2017.

[61] Y. Omote, T. Shinagawa, and K. Kato. Improving Agility and Elasticity in Bare-metal Clouds. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 145–159, New York, NY, USA, 2015. ACM.

[62] Openstack. Ironic. https://docs.openstack.org/ironic/latest/, 2018.

[63] Packet. The promise of the cloud delivered on bare metal. https://www.packet.net, 2017.

[64] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[65] Rackspace. Rackspace Cloud Big Data OnMetal. http://go.rackspace.com/baremetalbigdata/, 2015.

[66] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, 2016. USENIX Association.

[67] A. Regenscheid. Platform firmware resiliency guidelines. https://doi.org/10.6028/NIST.SP.800-193, May 2018.

[68] R. Ricci and t. E. Team. Precursors: Emulab. In R. McGeer, M. Berman, C. Elliott, and R. Ricci, editors, *The GENI Book*, pages 19–33. Springer International Publishing, Cham, 2016.

[69] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[70] J. Rutkowska. Intel x86 considered harmful, 2015. https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[71] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[72] N. Schear, P. T. Cable, II, T. M. Moyer, B. Richard, and R. Rudd. Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 65–77, New York, NY, USA, 2016. ACM.

[73] W. K. Sze, A. Srivastava, and R. Sekar. Hardening OpenStack Cloud Platforms against Compute Node Compromises. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*, pages 341–352, Xi'an, China, 2016. ACM Press.

[74] J. Szefer, P. Jamkhedkar, D. Perez-Botero, and R. B. Lee. Cyber defenses for physical attacks and insider threats in cloud computing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 519–524, New York, NY, USA, 2014. ACM.

[75] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. https://github.com/filebench/filebench/wiki, 2017.

[76] F. Tomonori and M. Christie. tgt: Framework for storage target drivers. In *Linux Symposium*, 2006.

[77] H. Wagner, D.-I. M. Zach, and D.-I. F. M. A.-P. Lintenhofer. BIOS-rootkit LightEater. 2015.

[78] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[79] R. Wojtczuk and J. Rutkowska. Attacking intel trusted execution technology. *Black Hat DC*, 2009.

[80] Y. A. Younis, K. Kifayat, and A. Hussain. Preventing and detecting cache side-channel attacks in cloud computing. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, pages 83:1–83:8. ACM, 2017.

[81] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud '10, 2010.

Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs

Gyusun Lee[†], Seokha Shin[*][†], Wonsuk Song[†], Tae Jun Ham[§], Jae W. Lee[§], Jinkyu Jeong[†]
[†]*Sungkyunkwan University,* [§]*Seoul National University*
{gyusun.lee, seokha.shin, wonsuk.song}@csi.skku.edu, {taejunham, jaewlee}@snu.ac.kr, jinkyu@skku.edu

Abstract

Today's ultra-low latency SSDs can deliver an I/O latency of sub-ten microseconds. With this dramatically shrunken device time, operations inside the kernel I/O stack, which were traditionally considered lightweight, are no longer a negligible portion. This motivates us to reexamine the storage I/O stack design and propose an *asynchronous* I/O stack (AIOS), where synchronous operations in the I/O path are replaced by asynchronous ones to overlap I/O-related CPU operations with device I/O. The asynchronous I/O stack leverages a lightweight block layer specialized for NVMe SSDs using the page cache without block I/O scheduling and merging, thereby reducing the sojourn time in the block layer. We prototype the proposed asynchronous I/O stack on the Linux kernel and evaluate it with various workloads. Synthetic FIO benchmarks demonstrate that the application-perceived I/O latency falls into single-digit microseconds for 4 KB random reads on Optane SSD, and the overall I/O latency is reduced by 15–33% across varying block sizes. This I/O latency reduction leads to a significant performance improvement of real-world applications as well: 11–44% IOPS increase on RocksDB and 15–30% throughput improvement on Filebench and OLTP workloads.

1 Introduction

With advances in non-volatile memory technologies, such as flash memory and phase-change memory, ultra-low latency solid-state drives (SSDs) have emerged to deliver extremely low latency and high bandwidth I/O performance. The state-of-the-art non-volatile memory express (NVMe) SSDs, such as Samsung Z-SSD [32], Intel Optane SSD [12] and Toshiba XL-Flash [25], provide sub-ten microseconds of I/O latency and up to 3.0 GB/s of I/O bandwidth [12, 25, 32]. With these ultra-low latency SSDs, the kernel I/O stack accounts for a large fraction in total I/O latency and is becoming a bottleneck to a greater extent in storage access.

[*]Currently at Samsung Electronics

One way to alleviate the I/O stack overhead is to allow user processes to directly access storage devices [6, 16, 27, 28, 49]. While this approach is effective in eliminating I/O stack overheads, it tosses many burdens to applications. For example, applications are required to have their own block management layers [49] or file systems [15, 43, 49] to build useful I/O primitives on top of a simple block-level interface (e.g., BlobFS in SPDK). Providing protections between multiple applications or users is also challenging [6, 16, 28, 43]. These burdens limit the applicability of user-level direct access to storage devices [49].

An alternative, more popular way to alleviate the I/O stack overhead is to optimize the kernel I/O stack. Traditionally, the operating system (OS) is in charge of managing storage and providing file abstractions to applications. To make the kernel more suitable for fast storage devices, many prior work proposed various solutions to reduce the I/O stack overheads. Examples of such prior work include the use of polling mechanism to avoid context switching overheads [5, 47], removal of bottom halves in interrupt handling [24, 35], proposal of scatter/scatter I/O commands [37, 50], simple block I/O scheduling [3, 24], and so on. These proposals are effective in reducing I/O stack overheads, and some of those are adopted by mainstream OS (e.g., I/O stack for NVMe SSDs in Linux).

In our work, we identify new unexplored opportunities to further optimize the I/O latency in storage access. The current I/O stack implementation requires many operations to service a single I/O request. For example, when an application issues a read I/O request, a page is allocated and indexed in a page cache [36]. Then, a DMA mapping is made and several auxiliary data structures (e.g., `bio`, `request`, `iod` in Linux) are allocated and manipulated. The issue here is that these operations occur synchronously before an actual I/O command is issued to the device. With ultra-low latency SSDs, the time it takes to execute these operations is comparable to the actual I/O data transfer time. In such case, overlapping those operations with the data transfer can substantially reduce the end-to-end I/O latency.

To this end, this paper proposes an *asynchronous* I/O

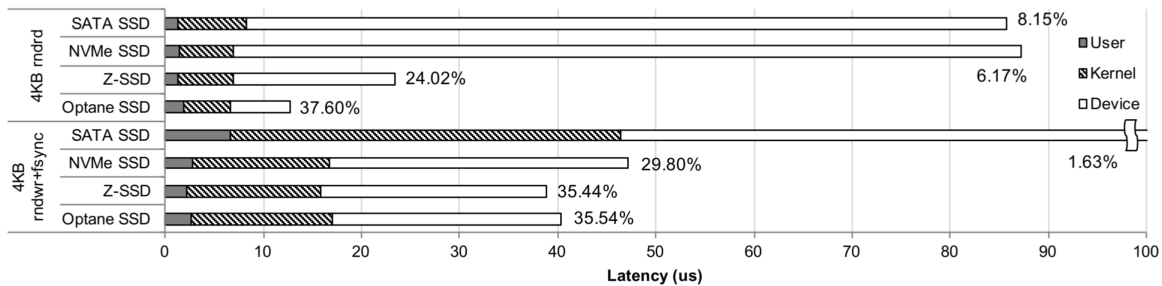


Figure 1: I/O latency and its breakdown with various storage devices. The numbers beside each bar denote the relative fraction of kernel time in the total I/O latency.

stack (AIOS), a low-latency I/O stack for ultra-low latency SSDs. Through a careful analysis of synchronous, hence latency-critical, system call implementations (i.e., `read()` and `fsync()`) in the Linux kernel, we identify I/O-related CPU operations that can be overlapped with device I/O operations and modify the Linux kernel to execute such CPU operations while the I/O device is processing a request. To further reduce the CPU overhead, we also introduce a lightweight block I/O layer (LBIO) specialized for NVMe-based SSDs, which enables the kernel to spend considerably less time in the block I/O layer. Our evaluation demonstrates that AIOS achieves up to 33% latency reduction for random reads and 31% latency reduction for random writes on FIO benchmarks [2]. Also, AIOS enables various real-world applications (e.g., a key-value store, database) to achieve higher throughput. Our contributions are summarized as follows:

- We provide a detailed analysis of the Linux kernel I/O stack operations and identify CPU operations that can overlap with device I/O operations (Section 2).
- We propose the lightweight block I/O layer (LBIO) specialized for modern NVMe-based SSD devices, which offers notably lower latency than the vanilla Linux kernel block layer (Section 3.1).
- We propose the asynchronous I/O stack for read and fsync paths in which CPU operations are overlapped with device I/O operations, thereby reducing the completion time of the read and fsync system calls (Section 3.2 and 3.3).
- We provide a detailed evaluation of the proposed schemes to show the latency reduction of up to 33% for random reads and 31% for random writes on FIO benchmarks [2] and substantial throughput increase on real-world workloads: 11–44% on RocksDB [10] and 15–30% on Filebench [40] and OLTP [18] workloads (Section 4).

2 Background and Motivation

2.1 ULL SSDs and I/O Stack Overheads

Storage performance is important in computer systems as data should be continuously supplied to a CPU to not stall the pipeline. Traditionally, storage devices have been much slower than CPUs, and this wide performance gap has existed for decades [26]. However, the recent introduction of modern storage devices is rapidly narrowing this gap. For example, today's ultra-low latency (ULL) NVMe SSDs, such as Samsung's Z-SSD [32], Intel's Optane SSD [12], and Toshiba's XL-Flash [25], can achieve sub-ten microseconds of I/O latency, which is orders of magnitude faster than that of traditional disks.

With such ultra-low latency SSDs, the kernel I/O stack [11, 38] no longer takes a negligible portion in the total I/O latency. Figure 1 shows the I/O latency and its breakdown for 4 KB random read and random write + `fsync` workloads on various SSDs[1]. The figure shows that ultra-low latency SSDs achieve substantially lower I/O latency than conventional SSDs. Specifically, their device I/O time is much lower than that of the conventional SSDs. On the other hand, the amount of time spent in the kernel does not change across different SSDs. As a result, the fraction of the time spent in the kernel becomes more substantial (i.e., up to 37.6% and 35.5% for the read and write workloads, respectively).

An I/O stack is composed of many layers [17]. A virtual file system (VFS) layer provides an abstraction of underlying file systems. A page cache layer provides caching of file data. A file system layer provides file system-specific implementations on top of the block storage. A block layer provides OS-level block request/response management and block I/O scheduling. Finally, a device driver handles device-specific I/O command submission and completion. In this paper, we target two latency-sensitive I/O paths (`read()` and `write()+fsync()`) in the Linux kernel and Ext4 file system with NVMe-based SSDs, since they are widely adopted system configurations from the mobile [13] to enterprise [33, 51].

2.2 Read Path

2.2.1 Vanilla Read Path Behavior

Figure 2 briefly describes the read path in the Linux kernel. Buffered read system calls (e.g., `read()` and `pread()` without `O_DIRECT`) fall into the VFS function (`buffered_read()` in the figure), which is an entry point to the page cache layer.

Page cache. Upon a cache miss (Line 3-4), the function

[1] The detailed evaluation configurations can be found in Section 4.1. Note that all the tested NVMe SSDs feature a non-volatile write cache, hence showing low write latency.

```
1  void buffered_read(file, begin, end, buf) {
2   for (idx=begin; idx<end; idx++) {
3    page = page_cache_lookup(idx)
4    if (!page) {
5     page_cache_sync_readahead(file, idx, end)
6     page = page_cache_lookup(idx)
7    } else {
8     page_cache_async_readahead(file, idx, end)
9    }
10   lock_page(page)
11   memcpy(buf, page, PAGE_SIZE)
12   buf += PAGE_SIZE
13  }
14 }
15
16 void page_cache_readahead(file, begin, end) {
17  init_list(pages)
18  for (idx=begin; idx<end; idx++) {
19   if (!page_cache_lookup(file, idx)) {
20    page = alloc_page()
21    page->idx = idx
22    push(pages, page)
23   }
24  }
25  readpages(file, pages)
26 }
27
28 void ext4_readpages(file, pages) {
29  blk_start_plug()
30  for (page : pages) {
31   add_to_page_cache(page, page->idx, file)
32   lba = ext4_map_blocks(file, page->idx)
33   bio = alloc_bio(page, lba)
34   submit_bio(bio)
35  }
36  blk_finish_plug()
37 }
```

Figure 2: Pseudocode for Linux kernel read path.

`page_cache_sync_readahead()` is called, in which missing file blocks are read into the page cache. It identifies all missing indices within the requested file range (Line 18-19), allocates pages and associates the pages with the missing indices (Line 20-21). Finally, it requests the file system to read the missing pages (Line 25).

File system. File systems have their own implementation of `readpages()`, but their behaviors are similar to each other. In Ext4 `ext4_readpages()` inserts each page into the page cache (Line 30-31), retrieves the logical block address (LBA) of the page (Line 32) and issues a block request to the underlying block layer (Line 33-34).

Linux batches several block requests issued by a thread in order to increase the efficiency of request handling in the underlying layers (also known as *queue plugging* [4]). When `blk_start_plug()` is called (Line 29), block requests are collected in a plug list of the current thread. When `blk_finish_plug()` (Line 36) is called or the current thread is context-switched, the collected requests are flushed to the block I/O scheduler.

After issuing an I/O request to a storage device, the thread rewinds its call stack and becomes blocked at the function `lock_page()` (Line 10). When the I/O request is completed,

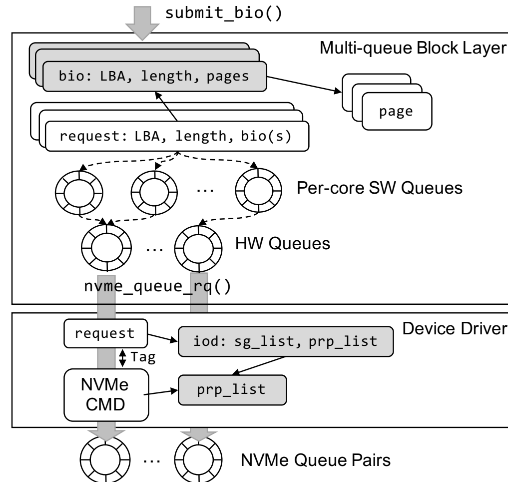


Figure 3: The overview of the multi-queue block layer. The shaded rectangles are dynamically allocated objects.

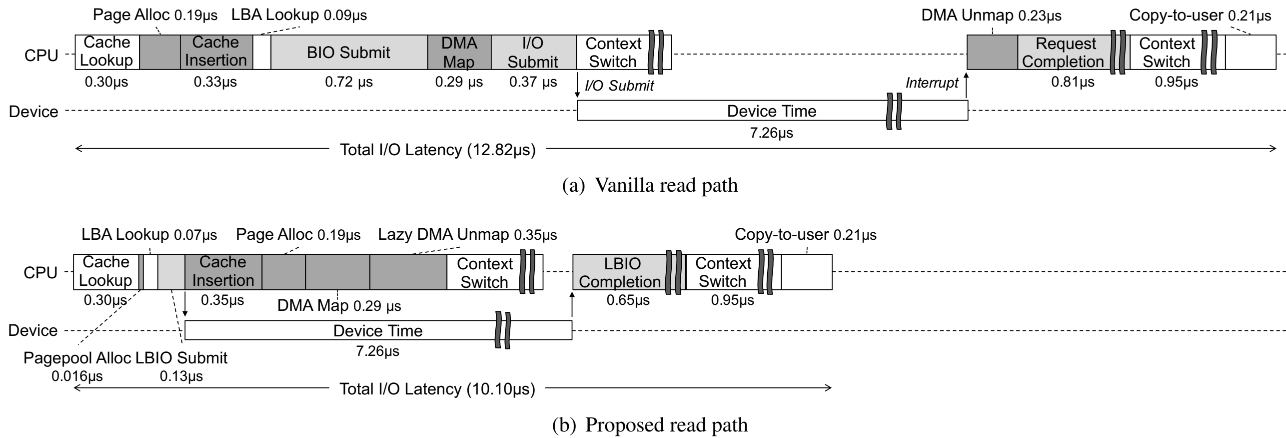
the interrupt handler releases the lock of the page, which wakes up the blocked thread. Finally, the cached data are copied to the user buffer (Line 11-12).

Block layer. Figure 3 shows the overview of the multi-queue block layer, which is the default block layer for NVMe SSDs in the Linux kernel, and the device driver layer. In the block layer, a `bio` object is allocated using a slab allocator and initialized to contain the information of a single block request (i.e., LBA, I/O size and pages to copy-in) (Line 33). Then, `submit_bio()` (Line 34) transforms the `bio` object to a `request` object and inserts the `request` object into request queues, where I/O merging and scheduling are performed [3, 8]. The `request` object passes through a per-core software queue (`ctx`) and hardware queue (`hctx`) and eventually reaches the device driver layer.

Device driver. A `request` is dispatched to the device driver using `nvme_queue_rq()`. It first allocates an `iod` object, a structure having a scatter/gather list, and uses it to perform DMA (direct memory access) mapping, hence allocating I/O virtual addresses (or DMA addresses) to the pages in the dispatched request. Then, a `prp_list`, which contains physical region pages (PRP) in the NVMe protocol, is allocated and filled with the allocated DMA addresses. Finally, an NVMe command is created using the `request` and the `prp_list` and issued to an NVMe submission queue. Upon I/O completion, the interrupt handler unmaps DMA addresses of the pages and calls a completion function, which eventually wakes up the blocked thread. While the above describes the basic operations in the read path, the roles of the block and device driver layers are identical in the write path.

2.2.2 Motivation for Asynchronous Read Path

Figure 4(a) summarizes the operations and their execution times in the read path explained in Section 2.2.1. The main problem is that a single read I/O path has many operations



(a) Vanilla read path

(b) Proposed read path

Figure 4: The operations and their execution times in the read paths during 4 KB random read on Optane SSD. (drawn to scale in time)

that occur synchronously to an actual device I/O operation. This synchronous design is common and intuitive and works well with slow storage devices because in such cases, the time spent on CPU is negligible compared to the total I/O latency. However, with ultra-low latency SSDs, the amount of CPU time spent on each operation becomes a significant portion of the total I/O latency.

Table 1 summarizes the ratio of each operation to the total kernel time. With small I/O sizes, the context switching is the most time-consuming operation. In this case, it is possible to reduce the overhead through the use of polling [5, 47]. The copy-to-user is another costly operation but is a necessary operation when file data is backed by the page cache. If we exclude these two operations, the remaining operations account for 45-50% of the kernel time.

Upon careful analysis of the code, we find that many of the remaining operations do not have to be performed before or after the device I/O time. In fact, such operations can be performed while the device I/O operation is happening since such operations are mostly independent of the device I/O operation. This motivates us to overlap such operations with the device I/O operation as sketched in Figure 4(b) (shaded in dark gray).

Layer	Action	Lines in Figure 2	% in kernel time
Page cache	Missing page lookup Page allocation	Line 18-24	9–10.8%
	Copy-to-user	Line 11-12	4.5–12%
File system	Page cache insertion LBA retrieval `bio` alloc/submit Make `request` from `bio`	Line 30-35	26–28.5%
Block	I/O scheduling (noop)	Line 36	10–11%
Driver	DMA mapping/unmapping NVMe command submit		
Scheduler	Context switch (2 times)	Line 10	25–41.5%

Table 1: Summary of operations and their fractions in kernel time in the read path. (4–16 KB FIO rndrd on Optane SSD)

2.3 Write Path

2.3.1 Vanilla Write Path Behavior

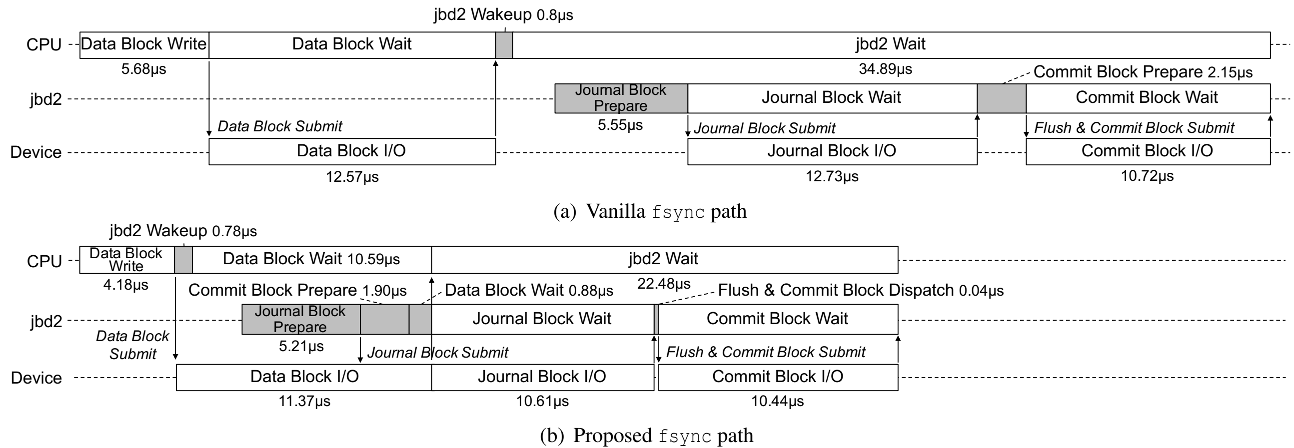
Buffered write system calls (e.g., `write()`) usually buffer the modified data in the page cache in memory. When an application calls `fsync()`, the kernel actually performs write I/Os to synchronize the dirtied data with a storage device.

The buffered write path has no opportunity to overlap computation with I/O because it does not perform any I/O operation. On the other hand, `fsync` accompanies several I/O operations due to the writeback of dirtied data as well as a crash consistency mechanism in a file system (e.g., file system journaling). Since `fsync` most heavily affects the application performance in the write path, we examine it in more detail.

Figure 5(a) shows the operations and their execution times during an `fsync` call on Ext4 file system using ordered journaling. First, an application thread issues write I/Os for dirty file blocks and waits for them to complete. Then, the application thread wakes up a journaling thread (`jbd2` in Ext4) to commit the file system transaction. It first prepares the write of modified metadata (*journal block* in the figure) onto the journal area and issues the write I/O. Then, it waits for the completion of the write. Once completed, it prepares the write of a commit block and issues the write I/O. A flush command is enforced between the journal block write and the commit block write to enforce the ordering of writes [45]. Hence, total three device I/O operations occur for a single `fsync` call.

2.3.2 Motivation for Asynchronous Write Path

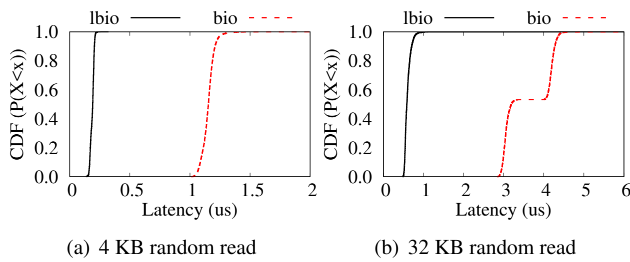
As in the case of the read path, there is also an opportunity to overlap the device I/O operations with the computation parts in the `fsync` path. As shown in Figure 5(a), the journaling thread performs I/O preparation and I/O waiting synchronously. Each I/O preparation includes assigning blocks in the journal area to write on, allocating buffer pages, allo-



(a) Vanilla `fsync` path

(b) Proposed `fsync` path

Figure 5: The operations and their execution times in the `fsync` paths during 4 KB random write with `fsync` on Optane SSD. (drawn to scale in time)



(a) 4 KB random read

(b) 32 KB random read

Figure 6: The CDF of the block I/O submission latency in the Linux block layer (*bio*) and the proposed lightweight block layer (*lbio*) on Optane SSD.

cating/submitting a `bio` object, assigning DMA address, and so forth. If these CPU operations are overlapped with the previous device I/O time, the total latency of the `fsync` system call can be greatly reduced as shown in Figure 5(b).

2.4 Motivation for Lightweight Block Layer

The Linux kernel uses the multi-queue block layer for NVMe SSDs by default to scale well with multiple command queues and multi-core CPUs [3]. This block layer provides functionality like block I/O submission/completion, request merging/reordering, I/O scheduling and I/O command tagging [3]. While these features are necessary for general block I/O management, they delay the submission time of an I/O command to a storage device.

Figure 6 shows the block I/O submission latency, time from allocating a `bio` object to dispatching an I/O command to a device (denoted as *bio*); the figure also includes the same measurement using the proposed lightweight block layer in Section 3.1 (denoted as *lbio*). We use two I/O sizes, 4 KB and 32 KB, to minimize and maximize the number of dynamic memory allocations during I/O submissions, respectively. Considering that the device time for a 4 KB read is around 7.3 μs on ultra-low latency SSDs, the amount of time spent in the block layer is about 15% of the device time, which is a non-negligible portion.

While block I/O submission/completion and I/O command tagging are necessary features, request merging/reordering and I/O scheduling are not significant. The multi-queue block layer supports various I/O schedulers [8] but its default configuration is *noop* since many studies report that I/O scheduling is ineffective for reducing I/O latency for latency-critical applications on fast storage devices [34, 46, 51]. I/O scheduling can also be replaced by the device-side I/O scheduling capability [14]. The effectiveness of request merging/reordering is also questionable in ultra-low latency SSDs because of their high random access performance and the low probability to find adjacent or identical block requests.

Based on these intuitions, we propose to simplify the roles of the block layer and make it specialized for our asynchronous I/O stack to minimize its I/O submission delay.

3 Asynchronous I/O Stack

The proposed asynchronous I/O stack (AIOS) consists of two components: the lightweight block I/O layer (LBIO) and the modified I/O stack that overlaps I/O-related computations with the device I/O operations. This section first explains LBIO and then explains the modified read and write paths.

3.1 Lightweight Block I/O Layer

To minimize the software overheads in the block layer, we design a lightweight block I/O layer (or LBIO), a scalable, lightweight alternative to the existing multi-queue block layer. LBIO is designed for low-latency NVMe SSDs and supports only I/O submission/completion and I/O command tagging. Figure 7 shows the overview of our proposed LBIO.

Unlike the original multi-queue block layer, LBIO uses a single memory object, `lbio`, to represent a single block I/O request, thereby eliminating the time-consuming `bio`-to-`request` transformation in the original block layer. Each

`lbio` object contains LBA, I/O length, pages to copy-in and DMA addresses of the pages. Containing DMA addresses in `lbio` leverages the asynchronous DMA mapping feature explained in the following sections. An `lbio` only supports 4 KB-aligned DMA address with I/O length of multiple sectors to simplify the codes initializing and submitting block I/O requests. This approach is viable with the assumption of using the page cache layer. Similar to the original block layer, LBIO supports queue plugging to batch multiple block I/O requests issued by a single thread.

LBIO has a global `lbio` two-dimensional array whose row is dedicated to each core, and a group of rows is assigned to each NVMe queue pair as shown in Figure 7. For example, if a system has 8 cores and 4 NVMe queue pairs, each `lbio` array row is one-to-one mapped to each core and two consecutive rows are mapped to an NVMe queue pair. When the number of NVMe queue pairs is equal to the number of cores, lockless `lbio` object allocations and NVMe command submissions are possible, as in the existing multi-queue block layer. The index of an `lbio` object in the global array is used as a tag in an NVMe command. This eliminates the time-consuming tag allocation in the original block layer.

Once an `lbio` is submitted, the thread directly calls `nvme_queue_lbio()` to dispatch an NVMe I/O command. Note that LBIO does not perform I/O merging or I/O scheduling, and thus reduces I/O submission delay significantly. Without the I/O merging, it is possible for two or more `lbio`'s to access the same logical block. This potentially happens in the read path and is resolved by the page cache layer (see Section 3.2). However, this does not happen in the write path because the page cache synchronizes writeback of dirty pages.

Figure 6 shows the reduced I/O submission latency with LBIO. On average, a block I/O request in LBIO takes only 0.18–0.60 μs, which is 83.4%–84.4% shorter latency compared to that of the original block layer.

3.2 Read Path

In addition to the introduction of LBIO, our approach reduces the I/O latency of the read path by detaching synchronous operations from the critical path as sketched in Figure 4(b). The following subsections describe each relocated operation and additional work to support the relocation.

3.2.1 Preloading Extent Tree

For a read operation, retrieving LBAs corresponding to the missing file blocks is a necessary step to issue a block request and thus this operation should be in the critical path. Instead of taking this step off the critical path, our proposal focuses on reducing its latency itself. The implementation of the Linux Ext4 file system caches logical to physical file block mappings in memory, and this cache is called extent status tree [19]. When a mapping can be found in the cache, obtaining an LBA takes a relatively short time; however, when the mapping is not found, the system has to issue an I/O request to read the missing mapping block and thus incurs much longer delay.

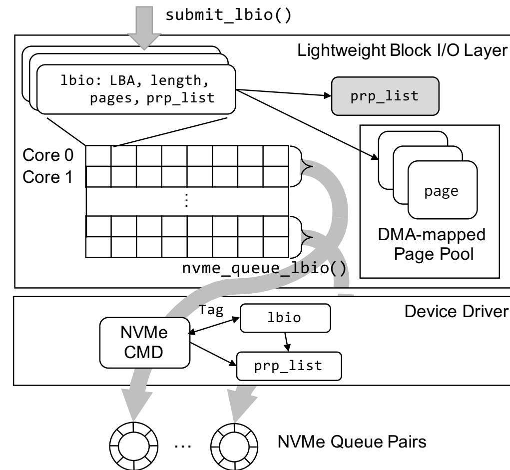


Figure 7: The proposed lightweight block I/O layer (LBIO). Shaded objects are dynamically allocated.

To avoid this unnecessary overhead, we adopt a plane separation approach [28]. In the control plane (e.g., file open), the entire mapping information is preloaded in memory. By doing so, the data plane (e.g., read and write) can avoid the latency delay caused by a mapping cache miss. The memory costs of caching an entire tree can be high; the worst case overhead is 0.03% of the file size in our evaluation. However, when there is little free memory, the extent cache evicts unlikely-used tree nodes to secure free memory [19]. To reduce the space overhead even further, this technique can be selectively applied to files requiring low-latency access.

3.2.2 Asynchronous Page Allocation/DMA Mapping

Preparation of free pages is another essential operation in the read path. In the original read path, a page allocator of the kernel performs this task, and it consumes many CPU cycles. For example, a single page allocation takes 0.19 μs on average in our system as shown in Figure 4(a). Similarly, assigning a DMA address to each page (DMA mapping) takes a large number of CPU cycles (754 cycles or 0.29 μs). Our approach is to take these operations off from the critical path and perform them while the device I/O operation is happening.

To this end, we maintain a small set of DMA-mapped free pages (a linked list of 4 KB DMA-mapped pages) for each core. With this structure, only a few memory instructions are necessary to get free pages from the pool (*Pagepool Alloc* in Figure 4(b)). The consumed pages are refilled by invoking page allocation and DMA mapping while the device I/O operation is occurring. This effectively hides the time for both page allocation and DMA mapping from the application-perceived I/O latency as shown in the figure. Note that, when the number of free pages in the pool is smaller than the read request size, page allocation and DMA mapping happens synchronously as in the vanilla kernel case.

3.2.3 Lazy Page Cache Indexing

Insertion of a page into a page cache index structure is another source of the large kernel I/O stack latency. Our approach is to overlap this operation with the device I/O operation while resolving the potentially duplicated I/O submissions.

In the vanilla kernel, the page cache works as a synchronization point that determines whether a block I/O request for a file can be issued or not. File blocks whose cache pages are successfully inserted into the page cache are allowed to make block requests (Line 31 in Figure 2), and a spinlock is used to protect the page cache from concurrent updates. Consequently, no duplicated I/O submission occurs for the same file block.

However, if we delay the page cache insertion operation to a point after submitting an I/O command to a device, it is possible for another thread to miss on the same file block and to issue a duplicate block request. To be exact, this happens if another thread accesses the page cache after the I/O request is submitted but before the page cache entry is updated.

Our solution is to allow duplicated block requests but resolve them at the request completion phase. Although there are multiple block requests associated with the same file block, only a single page is indexed in the page cache. Then, our scheme marks other pages as *abandoned*. The interrupt handler frees a page associated with the completed block request if it is marked *abandoned*.

3.2.4 Lazy DMA Unmapping

The last long-latency operation in the read path is DMA unmapping that occurs after the device I/O request is completed. The vanilla read path handles this in the interrupt handler, which is also in the critical path. Our scheme delays this operation to when a system is idle or waiting for another I/O request (*Lazy DMA unmap* in Figure 4(b)).

Note that this scheme prolongs the time window in which the DMA buffer is accessible by the storage device. This is essentially an extended version of the deferred protection scheme used in Linux by default [20]. Deferring the DMA unmapping (either in our scheme or in Linux) may potentially create a vulnerability window from a device-side DMA attack. However, with an assumption that the kernel and the device are neither malicious nor vulnerable, the deferred protection causes no problem [20]. If the assumption is not viable, users can disable the lazy DMA unmapping.

3.3 Write and fsync Path

As explained in Section 2.3.1, an `fsync` system call entails multiple I/O operations, and thus it is not possible to reuse the same scheme we proposed for the read path. For example, by the time `fsync` happens, pages are already allocated and indexed in the page cache. Instead of overlapping the I/O-related computation with individual device I/O operation, we focus on applying the overlapping idea to the entire file system journaling process.

Specifically, we overlap the computation parts in the journaling thread with the previous I/O operations in the same write path. As shown in Figure 5(a), there are two I/O preparation operations: journal block preparation and commit block preparation. Each preparation operation includes allocating buffer pages, allocating a block on the journal area, calculating the checksum and computation operations within the block and device driver layers. Since these operations only modify in-memory data structures, they have no dependency on the previous I/O operation in the same write path. Note that, at any given time, only a single file system transaction can be committed. While a transaction is in the middle of commit, no other file system changes can be entangled to the current transaction being committed. Hence, if the ordering constraint, which allows the write of a commit block only after the data blocks and journal blocks are made durable on the storage media, is guaranteed, our approach can provide the same crash consistency semantic provided by the vanilla write path.

To this end, we change the `fsync` path as shown in Figure 5(b). Upon an `fsync` system call, an application thread issues the writeback of dirty data pages first. Then, it wakes up the journaling thread in advance to overlap the data block I/Os with the computation parts in the journaling thread. The application thread finally waits for the completion of the writeback I/Os as well as the completion of the journal commit. While the data block I/O operations are happening, the journaling thread prepares the journal block writes and issues their write I/Os. Then, it prepares the commit block write and waits for the completion of all the previous I/O operations associated with the current transaction. Once completed, it sends a flush command to the storage device to make all the previous I/Os durable and finally issues a write I/O of the commit block using a write-through I/O command (e.g., FUA in SATA). After finishing the commit block write, the journaling thread finally wakes up the application thread.

When an `fsync` call does not entail a file system transaction, it is not possible to overlap computation with I/O operation. In this case, the use of LBIO reduces its I/O latency.

3.4 Implementation

The proposed scheme is implemented in the Linux kernel version 5.0.5. A new file open flag, `O_AIOS`, is introduced to use the proposed I/O stack selectively. The current implementation supports `read()`, `pread()`, `fsync()`, and `fdatasync()` system calls. For now, other asynchronous or direct I/O APIs are not supported.

The LBIO layer shares the NVMe queue pairs used in the original block layer. The spinlock of each queue pair provides mutual exclusion between LBIO and the original block layer. The most significant bit of a 16-bit tag is reserved to distin-

	Object	Linux block	LBIO
128 KB block request	`(l)bio` + `(l)bio_vec`	648 bytes	704 bytes
	`request`	384 bytes	
	`iod`	974 bytes	
	`prp_list`	4096 bytes	4096 bytes
	Total	6104 bytes	4800 bytes
Statically allocated (per-core)	`request` pool	412 KB	
	`lbio` array		192 KB
	free page pool		256 KB

Table 2: Memory cost comparison

Server	Dell R730	
OS	Ubuntu 16.04.4	
Base kernel	Linux 5.0.5	
CPU	Intel Xeon E5-2640v3 2.6 GHz 8 cores	
Memory	DDR4 32 GB	
Storage devices	*Z-SSD*	Samsung SZ985 800 GB
	Optane SSD	Intel Optane 905P 960 GB
	NVMe SSD	Samsung PM1725 1.6 TB
	SATA SSD	Samsung 860 Pro 512 GB

Table 3: Experimental configuration

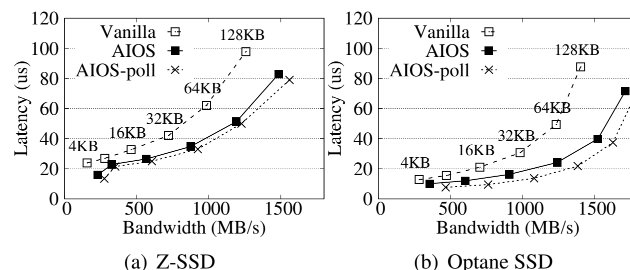


Figure 8: FIO single-thread random read latency and throughput with varying block sizes.

guish the two block layers. Since the NVMe SSDs used for evaluation supports 512 or 1024 entries for each queue, the remaining 15 bits are sufficient for command tagging.

Table 2 summarizes the memory cost of our scheme and the original block layer. To support a single 128 KB block request, both layers use a comparable amount of memory for `(l)bio`, `(l)bio_vec`, and `prp_list` objects. However, the original block layer requires two additional memory objects: `request` and `iod`, and thus requires extra memory compared to LBIO.

As for the cost of statically allocated memory, the original block layer maintains a pool of `request` objects (1024 objects with 1024 I/O queue depth), which requires 412 KB memory per core. LBIO replaces the per-core request pool with the per-core `lbio` row of size 192 KB. Meanwhile, our scheme also maintains a pool of DMA-mapped free pages. We maintain 64 free pages for each free page pool, hence consuming additional 256 KB memory per core.

In order to implement the AIOS `fsync` path, the `jbd2` wakeup routine in `ext4_sync_file()` is relocated to the position between data block write and data block wait. The function `jbd2_journal_commit_transaction()` is also modified to implement our scheme. The routine to prepare a commit block is moved to the position before the waiting routine for the journal block writes. The waiting routine for data block writes (using `t_inode_list`) is also relocated to the position before the waiting for journal block writes. The routine to issue commit block write I/O (i.e., `submit_bh()` in the vanilla path) is split into two routines: one for allocating an `lbio` and mapping DMA address, and the other for submitting an I/O command to a device (i.e., `nvme_queue_lbio()`). With this separation, AIOS can control the time to submit the commit block I/O so that it can satisfy the ordering constraints, while allowing the overlap of block request-related computations (e.g., DMA mapping) with the previous I/O operations.

4 Evaluation

4.1 Methodology

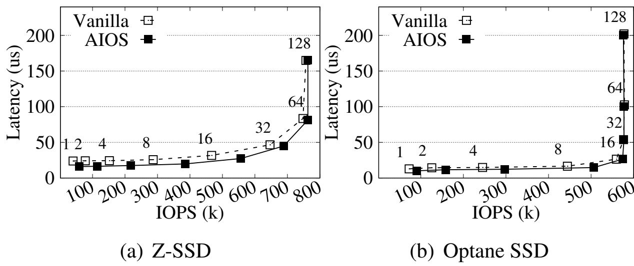
We use Dell R730 Server machine with Intel Xeon E5-2640 CPU and 32 GB DDR4 memory for our experiments. For the ultra-low latency storage devices, we evaluate both Samsung Z-SSD and Intel Optane SSD; both integrate a non-volatile write cache, which ignores flush and FUA commands in the block layer. We implement our proposed scheme AIOS on Linux kernel 5.0.5, denoted as *AIOS*. The baseline is the vanilla Linux kernel 5.0.5 using the Ext4 file system, denoted as *vanilla*. Table 3 summarizes our experimental setup.

For evaluation, we utilize both the synthetic microbenchmark and real-world workloads. For the synthetic microbenchmark, we use FIO [2] using the *sync* engine with varying I/O request sizes, I/O types, the number of threads and so forth. For real-world workloads we utilize various applications such as key-value store (RocksDB [10]), file-system benchmark (Filebench-varmail [40]), and OLTP workload (Sysbench-OLTP-insert [18] on MySQL [22]). Specifically, we run *readrandom* and *fillsync* workloads of the DBbench [30] on RocksDB; each representing a read-intensive case and `fdatasync`-intensive case, respectively. Filebench-varmail is `fsync`-intensive, and Sysbench-OLTP-insert is `fdatasync`-intensive.

4.2 Microbenchmark

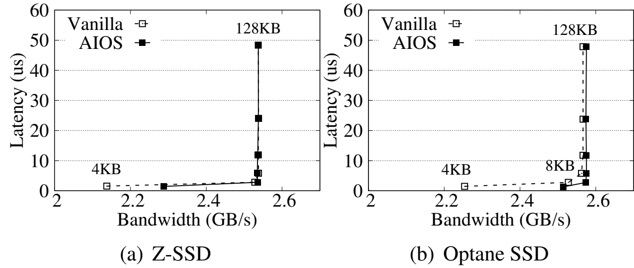
4.2.1 Read Performance

Random read latency. Figure 8 shows the effect of AIOS on FIO random read latency and throughput with varying block sizes. The figure shows that AIOS reduces random read latency by 15–33% when compared to the vanilla kernel on both Z-SSD and Optane SSD. In general, a larger block size results in greater latency reduction because a larger portion of the read-related kernel computation gets overlapped with the device I/O operation (see Figure 4). One important note is that AIOS achieves single-digit microseconds latency for a 4 KB random read on Optane SSD, which was previously not possible due to substantial read path overheads.



(a) Z-SSD (b) Optane SSD

Figure 9: FIO 4 KB random read latency and throughput (in IOPS) with varying the number of threads.



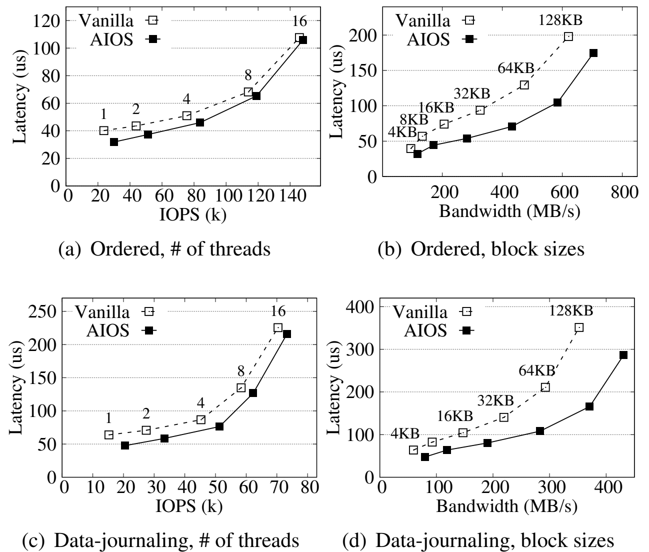
(a) Z-SSD (b) Optane SSD

Figure 10: Single-thread FIO sequential read latency and bandwidth with varying block sizes.

Polling vs. interrupt. Figure 8 also shows the impact of I/O completion scheme to the read latency. In the figure, *AIOS-poll* denotes the AIOS scheme using not interrupt but polling as its I/O completion method. In general, polling is better than interrupt in terms of latency because it eliminates context switches [47]; Table 1 has shown that context switches account for the largest fraction in kernel time. With small I/O sizes, the latency reduction from polling is comparable to that from AIOS. However, with large I/O sizes, the latency reduction from AIOS is greater than that from polling because of I/O-computation overlap to a greater extent. Please note that the interrupt mode is used by default in the rest of this section.

Random read throughput. Figure 9 shows the FIO 4 KB random read latency and throughput in I/O operations per second (IOPS) with varying the number of threads. Here, each thread is set to issue a single I/O request at a time (i.e., queue depth is one). In this setting, a large number of threads means that a large number of I/O requests are outstanding, hence mimicking high queue-depth I/O. As shown in the figure, both Z-SSD and Optane SSD achieve notably higher IOPS (i.e., up to 47.1% on Z-SSD and 26.3% on Optane SSD) than the baseline when the number of threads is less than 64. From that point, the device bandwidth gets saturated, and thus AIOS does not result in additional performance improvement.

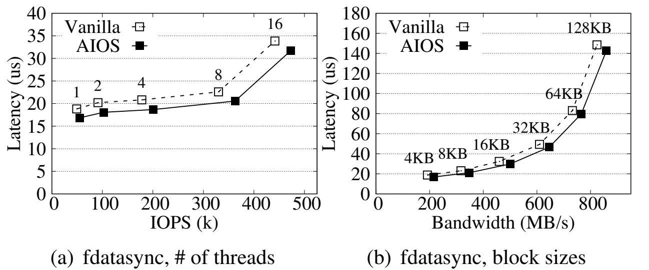
Sequential read bandwidth. Figure 10 shows the effect of AIOS on a single-thread sequential read workload with varying block sizes. Because the workload uses buffered reads, the readahead mechanism in Linux prefetches data blocks with large block size (128 KB) into the page cache. This results in high sustained bandwidth. In both Z-SSD and Optane SSD



(a) Ordered, # of threads (b) Ordered, block sizes

(c) Data-journaling, # of threads (d) Data-journaling, block sizes

Figure 11: Fsync performance with different journaling modes, number of threads and block sizes on Optane SSD.



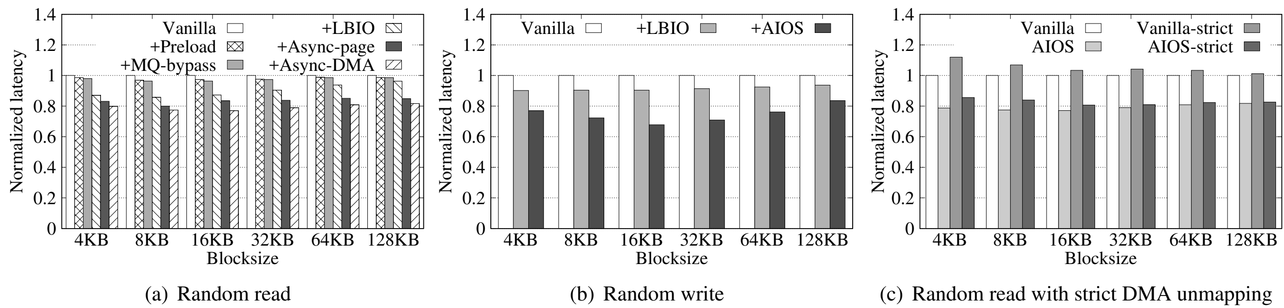
(a) fdatasync, # of threads (b) fdatasync, block sizes

Figure 12: Fdatasync performance with ordered mode with varying the number of threads and block sizes on Optane SSD.

cases, AIOS enables higher sustained bandwidth usage only for 4 KB block reads. For 16–128 KB blocks, AIOS does not result in any bandwidth improvement because the baseline scheme already reaches peak bandwidth.

4.2.2 Write Performance

Fsync performance. Figure 11 shows the performance impact of AIOS on FIO 4 KB random write followed by fsync workload. Here, we evaluate two journaling modes: ordered mode and data-journaling mode. With a single thread, our scheme shows IOPS improvement by up to 27% and 34% and latency reduction by up to 21% and 26% in the ordered mode and data journaling mode, respectively. With increasing the number of threads, natural overlap between computation and I/O occurs, thereby diminishing the performance benefit of our scheme. With large block sizes, the long I/O-computation overlapping happens, thereby widening the absolute performance gap between our scheme and the baseline. In the data-journaling mode, the length of the overlapped portion is longer than that of the ordered mode, and thus its latency advantage



(a) Random read (b) Random write (c) Random read with strict DMA unmapping

Figure 13: Normalized Latency of FIO 4 KB random read or write workloads with varying optimization levels on Optane SSD.

is slightly larger than that of the ordered mode.

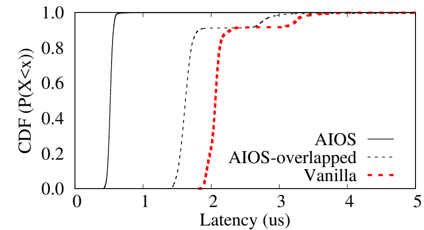
Fdatasync performance. Figure 12 shows the performance impact of AIOS on FIO 4 KB random write followed by fdatasync under the ordered mode. Unlike fsync, fdatasync does not journal metadata if the metadata has not changed; hence showing fewer performance gains than the fsync cases. Our AIOS presents up to 12% IOPS increase with a single thread. AIOS shows up to 10.5% latency decrease on the 4 KB random write workload using a single thread.

4.2.3 Performance Analysis

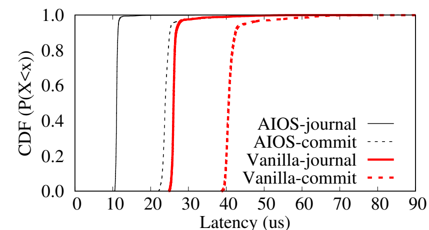
Read latency. Figure 13(a) compares the 4 KB random read latency on Optane SSD with varying optimization levels. The leftmost bar represents the baseline, and the next five bars represent the latency with cumulatively applying the optimizations presented throughout the paper. First, the preloading of extent tree (*+Preload*, Section 3.2.1) shows up to 3.1% latency reduction. Second, bypassing the block multi-queue scheduling (*+MQ-bypass*) provides 1.1% of additional latency reduction. The complete use of LBIO (*+LBIO*, Section 3.1) reduces I/O latency additionally by up to 12.5% because of its low overhead. By overlapping only cache indexing (Section 3.2.3) and page allocation (Section 3.2.2) with device I/O operation, the latency is reduced by up to 11.9% (*+Async-page*). Finally, the asynchronous DMA mapping/unmapping (*+Async-DMA*, Section 3.2.4) further reduces the I/O latency by up to 7.8%. The latency benefit of LBIO is similar across different block sizes. The benefit of the asynchronous operations, however, increases as the block size increases.

Write latency. Figure 13(b) compares the 4 KB random write+fsync latency on Optane SSD with varying optimization levels. For the write operation, the use of LBIO (*+LBIO*) shows up to 9.8% of latency reduction. Our asynchronous write path (*+AIOS*, Section 3.3) achieves additional latency reduction by up to 24.4%.

Cost of safety. Figure 13(c) shows the latency penalty of disabling the deferred DMA unmapping (i.e., unmapping DMA addresses immediately after I/O completion). *AIOS-strict* denotes our scheme without the lazy DMA unmapping (Section 3.2.4). For a fair comparison, we also measured the performance of the baseline without the deferred DMA unmapping [20] (denoted as *Vanilla-strict*). As shown in the figure, the use of strict DMA unmapping incurs a slight increase in I/O latency for both schemes by 1.2–11.9%.



(a) Latency from `read` entry to I/O cmd. submit

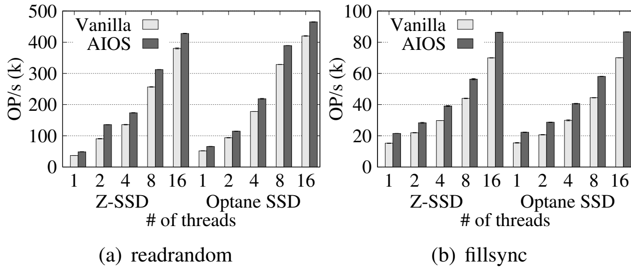


(b) Latency from `fsync` entry to each I/O cmd. submit (journal block and commit block)

Figure 14: CDF of I/O command submission latencies in 4 KB random read and 4 KB random write+fsync workloads on Optane SSD.

Overlapping analysis. One of our key ideas is to overlap computations with I/O so that the I/O command can be submitted as early as possible. To clarify this behavior, we measured the latency to submit I/O command(s) to a storage device and present the cumulative distribution function (CDF) of latencies. Figure 14(a) shows the time between the `read` system call entry and the I/O command submission. We also show the time to complete the overlapped operations in our scheme (denoted as *AIOS-overlapped*). As shown in the figure, the I/O submission latency is greatly reduced to 1.63 μs on average (75% reduction compared to the baseline). Also, note that the time to complete the overlapped operations is earlier than the I/O submission latency in the baseline. This is because of the additional latency reduction achieved by LBIO.

Similar measurement is also made on the write paths. Figure 14(b) shows the time between the `fsync` system call entry and the I/O command submission for journal block(s) and commit block (denoted as *-journal* and *-commit*, respectively).



(a) readrandom (b) fillsync

Figure 15: DBbench on RocksDB Performance (IOPS) with varying the number of threads.

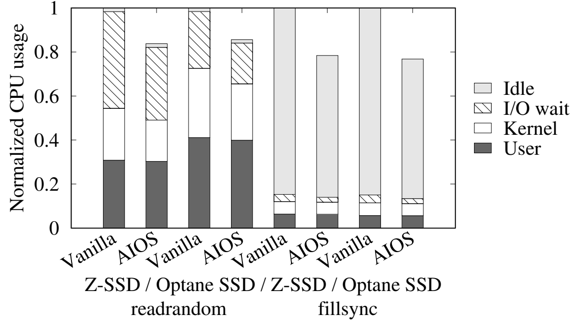


Figure 16: Normalized CPU usage breakdown for DBbench (*readrandom* and *fillsync*) on RocksDB using eight threads.

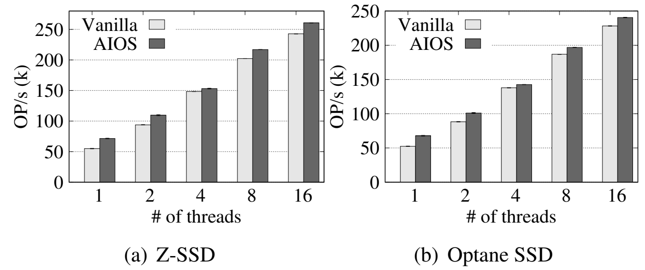
As shown in the figure, our scheme brings the I/O command submission times forward. Interestingly, the time to submit commit block I/O in AIOS is even earlier than the time to submit journal block I/O in the baseline.

4.3 Real-world Applications

4.3.1 Key-value Store

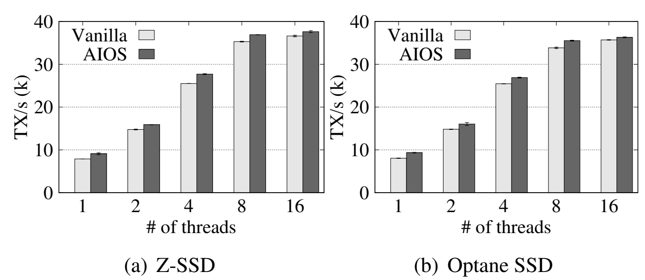
Performance. Figure 15 demonstrates the performance result of DBbench on RocksDB. Specifically, we evaluate the *readrandom* (64 GB dataset, 16-byte key and 1000-byte value) and *fillsync* (16 GB dataset) workloads in DBbench, each representing the random read-dominant case and the random write (and `fdatasync`)-dominant case. Overall, AIOS demonstrates notable speedups on the readrandom workload by 11–32% and the fillsync workload by 22–44%. Recall that AIOS allows duplicated I/Os because of the lazy page cache indexing feature (Section 3.2.3). Duplicated I/Os happen in this experiment. However, the frequency of such events is extremely low (e.g., less than once in 10 million I/Os on the readrandom workload).

CPU usage breakdown. Figure 16 shows the normalized CPU usage breakdown analysis of each workload using eight threads. Overall, AIOS reduces the CPU time spent on I/O wait and kernel. This indicates that AIOS effectively overlaps I/O operation with kernel operations without incurring extra overhead and LBIO effectively reduces the block I/O management overhead. Furthermore, by providing a low random read and write latency, AIOS reduces the overall runtime as well. The trend is similar in both Z-SSD and Optane SSD.



(a) Z-SSD (b) Optane SSD

Figure 17: Filebench-varmail performance with varying the number of threads.



(a) Z-SSD (b) Optane SSD

Figure 18: Sysbench-OLTP-insert performance with varying the number of threads.

4.3.2 Storage Benchmark and OLTP Workload

Figure 17 and Figure 18 show the Filebench-varmail (default configuration) and Sysbench-OLTP-insert (10 GB DB table size) performance with varying the number of threads. On the single thread cases of Filebench-varmail, utilizing AIOS results in 29.9% and 29.4% of improved throughput on Z-SSD and Optane SSD, respectively. Similarly, on the single thread cases of Sysbench-OLTP-insert, AIOS achieves 15.4% and 16.2% performance increase with Z-SSD and Optane SSD, respectively. In general, the use of multi-threading diminishes the benefits of our scheme because of the natural overlap of computation with I/O happens.

5 Related Work

Many prior works aim to alleviate the overheads of the kernel I/O stack, some of which are deployed in commodity OS kernels (e.g., Linux). Multi-context I/O paths can increase the I/O latency due to the overhead of context switching [5, 35]. Today's I/O path design for NVMe SSDs reduces this overhead by eliminating the bottom half of interrupt handling [24]. Using polling instead of interrupts is another solution for removing context switching from the I/O path [5, 47]. Hybrid polling is also proposed to reduce high CPU overheads [9, 21]. Simplified scheduling (e.g., Noop) is effective for reducing I/O latency in flash-based SSDs due to its high-performance random access [46, 51]. Instead of providing I/O scheduling in software, the NVMe protocol supports I/O scheduling on the device side in hardware [14, 24]. Support for differentiated I/O path was introduced to minimize the overhead of I/O path

for high priority tasks [5, 17, 48, 51], which is similar to our LBIO. However, to the best of our knowledge, there is no work applying the asynchronous I/O concept to the storage I/O stack itself.

There are proposals to change the storage interface for I/O latency reduction. Scatter/scatter I/O coalesces multiple I/O requests into a single command, thereby reducing the number of round trips in storage access [37, 42, 50]. DC-express attempts to minimize protocol-level latency overheads by removing doorbells and completion signals [44].

Improving the performance of `fsync` operation is important in many applications as it provides data durability. Nightingale et al. propose to extend the time to preserve data durability from the return of an `fsync` call to the time when the response is sent back to the requester (e.g., remote node) [23]. Using a checksum in the journal commit record can be effective in overlapping journal writes and data block writes [29], albeit checksum collision can become problematic in production systems [41]. OptFS [7] and BFS [45] propose write order-preserving system calls (`osync` and `fbarrier`). With the order-preserving system calls, the overlapping effect in the `fsync` path will be identical. However, when applications need the `fsync` semantic, operations occur synchronously with regard to I/Os.

User-level direct access can eliminate the kernel I/O stack overhead in storage access [6, 16, 28, 49]. The lack of a file system can be augmented by many different approaches from a simple mapping layer [28, 49] to user-level file systems [15, 43, 49]. However, enforcing isolation or protection between multiple users or processes should be carefully addressed [6], and hardware-level support is highly valuable [24]. However, this is not available at the moment.

6 Discussion and Future Work

I/O scheduling. I/O scheduling is necessary for certain computing domains (e.g., cloud computing) [1]. Early versions of the block multi-queue layer provided no I/O scheduling capability [3], but recently, several I/O schedulers have been integrated [8]. Our LBIO eliminates software-level request queues, and thus the current implementation is not compatible with software-level block I/O schedulers. However, the NVMe protocol can support device-side I/O scheduling (e.g., weighted round robin with urgent priority feature [14, 24]), which can augment LBIO. Furthermore, we believe that LBIO can support proper process/thread/cgroup-level I/O scheduling if we relax the static mapping between cores and NVMe queues. We leave this as future work.

File system coverage. Our current implementation is based on the Linux kernel with the Ext4 file system. However, we believe that other journaling file systems (e.g., XFS [39]) or copy-on-write file systems (e.g., Btrfs [31]) may provide similar opportunities for overlapping computation in the I/O path with device access, considering out-of-place updates

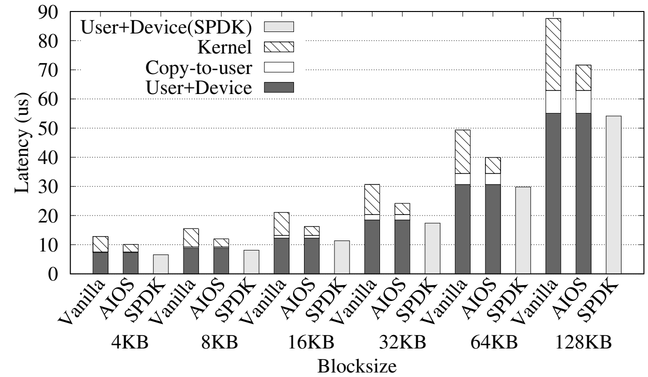


Figure 19: FIO random read latency breakdown in comparison with Intel SPDK on Optane SSD.

employed by these file systems.

Copy-to-user cost. AIOS greatly reduces the I/O stack overhead of the vanilla Linux kernel as shown in Figure 19. However, our proposal does not optimize copy-to-user operations, which remain as a non-negligible source of the overhead, especially when the requested block size is large. Although the in-memory copy is inevitable for buffered reads, we are seeking solutions to take off the memory copy from the critical path so that our proposal can compete with the user-level direct access approach.

7 Conclusion

We propose AIOS, an asynchronous kernel I/O stack customized for ultra-low latency SSDs. Unlike the traditional block layer, the lightweight block I/O (LBIO) layer of AIOS eliminates unnecessary components to minimize the delay in submitting I/O requests. AIOS also replaces synchronous operations in the I/O path with asynchronous ones to overlap computation associated with `read` and `fsync` with device I/O access. As a result, AIOS achieves single-digit microseconds I/O latency on Optane SSD, which was not possible due to high I/O stack overhead. Furthermore, AIOS demonstrates significant latency reduction and performance improvement with both synthetic and real-world workloads on Z-SSD and Optane SSD[2].

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Youjip Won, for their valuable comments. We also thank Prof. Jin-Soo Kim for his devotion of time at LAX and valuable technical feedback. This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1C1B2007273, NRF-2016M3C4A7952587) and by Samsung Electronics.

[2]The source code is available at https://github.com/skkucsl/aios.

References

[1] AHN, S., LA, K., AND KIM, J. Improving I/O resource sharing of linux cgroup for NVMe SSDs on multi-core systems. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)* (Denver, CO, USA, 2016).

[2] AXBOE, J. FIO: Flexible I/O tester. https://github.com/axboe/fio.

[3] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *International Systems and Storage Conference (SYSTOR '13)* (New York, NY, USA, 2013), pp. 22:1–22:10.

[4] BROWN, N. A block layer introduction part 1: the bio layer, 2017. https://lwn.net/Articles/736534/.

[5] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)* (Atlanta, GA, USA, 2010), pp. 385–395.

[6] CAULFIELD, A. M., MOLLOW, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)* (New York, NY, USA, 2012), pp. 387–400.

[7] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, USA, 2013), pp. 228–243.

[8] CORBET, J. Two new block I/O schedulers for 4.12, 2017. https://lwn.net/Articles/720675/.

[9] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing DRAM footprint with NVM in facebook. In *European Conference on Computer Systems (EuroSys '18)* (New York, NY, USA, 2018), pp. 42:1–42:13.

[10] FACEBOOK. Rocksdb. https://github.com/facebook/rocksdb/.

[11] HUFFMAN, A. Delivering the full potential of PCIe storage. In *IEEE Hot Chips Symposium* (2013), pp. 1–24.

[12] INTEL. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[13] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference (USENIX ATC '13)* (San Jose, CA, USA, 2013), pp. 309–320.

[14] JOSHI, K., YADAV, K., AND CHOUDHARY, P. Enabling NVMe WRR support in Linux block layer. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)* (Santa Clara, CA, USA, 2017).

[15] KIM, H.-J., AND KIM, J.-S. A user-space storage I/O framework for NVMe SSDs in mobile smart devices. *IEEE Transactions on Consumer Electronics 63*, 1 (2017), 28–35.

[16] KIM, H.-J., LEE, Y.-S., AND KIM, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)* (Denver, CO, USA, 2016).

[17] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the I/O path: A holistic approach for application performance. In *USENIX Conference on File and Storage Technologies (FAST '17)* (Santa Clara, CA, USA, 2017), pp. 345–358.

[18] KOPYTOV, A. Sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[19] KÁRA, J. Ext4 filesystem scaling. https://events.static.linuxfound.org/sites/events/files/slides/ext4-scaling.pdf.

[20] MARKUZE, A., MORRISON, A., AND TSAFRIR, D. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)* (New York, NY, USA, 2016), pp. 249–262.

[21] MOAL, D. L. I/O latency optimization with polling, 2017.

[22] MYSQL AB. MySQL. https://www.mysql.com.

[23] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Symposium on Operating Systems Design and Implementation (OSDI '06)* (Berkeley, CA, USA, 2006), pp. 1–14.

[24] NVM EXPRESS. NVM express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3c-2018.05.24-Ratified.pdf.

[25] OHSHIMA, S. Scaling flash technology to meet application demands, 2018. Keynote 3 at Flash Memory Summit 2018.

[26] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[27] PETER, S., LI, J., ZHANG, I., PORTS, D. R., ANDERSON, T., KRISHNAMURTHY, A., ZBIKOWSKI, M., AND WOOS, D. Towards high-performance application-level storage management. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)* (Philadelphia, PA, USA, 2014).

[28] PETER, S., LI, J., ZHANG, I., PORTS, D. R., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, USA, 2014), pp. 1–16.

[29] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *ACM Symposium on Operating Systems Principles (SOSP '05)* (New York, NY, USA, 2005), pp. 206–220.

[30] REECE, A. DBbench. https://github.com/memsql/dbbench.

[31] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS '13) 9*, 3 (2013), 9:1–9:32.

[32] SAMSUNG. Ultra-low latency with Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[33] SANDEEN, E. Enterprise filesystems, 2017. http://people.redhat.com/mskinner/rhug/q4.2017/Sandeen_Talk_2017.pdf.

[34] SAXENA, M., AND SWIFT, M. M. FlashVM: Virtual memory management on flash. In *USENIX Annual Technical Conference (USENIX ATC '10)* (Berkeley, CA, USA, 2010), pp. 14–14.

[35] SHIN, W., CHEN, Q., OH, M., EOM, H., AND YEOM, H. Y. OS I/O path optimizations for flash solid-state drives. In *USENIX Annual Technical Conference (USENIX ATC '14)* (Philadelphia, PA, USA, 2014), pp. 483–488.

[36] SILVERS, C. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *USENIX Annual Technical Conference (USENIX ATC '00)* (San Diego, CA, USA, 2000), pp. 285–290.

[37] SON, Y., HAN, H., AND YEOM, H. Y. Optimizing file systems for fast storage devices. In *ACM International Systems and Storage Conference (SYSTOR '15)* (New York, NY, USA, 2015), pp. 8:1–8:6.

[38] SWANSON, S., AND CAULFIELD, A. M. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer 46*, 8 (August 2013), 52–59.

[39] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *USENIX Annual Technical Conference (USENIX ATC '96)* (San Diego, CA, USA, 1996), vol. 15.

[40] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine 41*, 1 (2016), 6–12.

[41] TS'O, T. What to do when the journal checksum is incorrect, 2008. https://lwn.net/Articles/284038/.

[42] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *ACM Symposium on Cloud Computing (SoCC '12)* (New York, NY, USA, 2012), pp. 8:1–8:13.

[43] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *European Conference on Computer Systems (EuroSys '14)* (New York, NY, USA, 2014), pp. 14:1–14:14.

[44] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., MOAL, D. L., BUNKER, T., XU, J., SWANSON, S., AND BANDIĆ, Z. DC express: Shortest latency protocol for reading phase change memory over PCI express. In *USENIX Conference on File and Storage Technologies (FAST '14)* (Santa Clara, CA, USA, 2014), pp. 309–315.

[45] WON, Y., JUNG, J., CHOI, G., OH, J., SON, S., HWANG, J., AND CHO, S. Barrier-enabled IO stack for flash storage. In *USENIX Conference on File and Storage Technologies (FAST '18)* (Oakland, CA, USA, 2018), pp. 211–226.

[46] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of NVMe SSDs and their implication on real world databases. In *ACM International Systems and Storage Conference (SYSTOR '15)* (New York, NY, USA, 2015), pp. 6:1–6:11.

[47] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *USENIX conference on File and Storage Technologies (FAST '12)* (San Jose, CA, USA, 2012), pp. 3–3.

[48] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First class support for interactivity in commodity operating systems. In *USENIX Conference on Operating Systems Design and Implementation (OSDI '08)* (Berkeley, CA, USA, 2008), pp. 73–86.

[49] YANG, Z., HARRIS, J. R., WALKER, B., VERKAMP, D., LIU, C., CHANG, C., CAO, G., STERN, J., VERMA, V., AND PAUL, L. E. SPDK: A development kit to build high performance storage applications. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom '17)* (Hong Kong, 2017), pp. 154–161.

[50] YU, Y. J., SHIN, D. I., SHIN, W., YOUNG SONG, N., CHOI, J. W., KIM, H. S., EOM, H., AND EOM, H. Y. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Compueter Systems (TOCS '14) 32* (2014).

[51] ZHANG, J., KWON, M., GOUK, D., KOH, S., LEE, C., ALIAN, M., CHUN, M., KANDEMIR, M. T., KIM, N. S., KIM, J., AND JUNG, M. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *Symposium on Operating Systems Design and Implementation (OSDI '18)* (Carlsbad, CA, USA, 2018), pp. 477–492.

M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication

Nils Asmussen[†] [‡] Michael Roitzsch[†] [‡] Hermann Härtig[†] [‡]
[†] *Technische Universität Dresden, Germany* [‡] *Barkhausen Institut, Dresden, Germany*

Abstract

Performance and efficiency requirements are driving a trend towards specialized accelerators in both datacenters and embedded devices. In order to cut down communication overheads, system components are pinned to cores and fast-path communication between them is established. These fast paths reduce latency by avoiding indirections through the operating system. However, we see three roadblocks that can impede further gains: First, accelerators today need to be assisted by a general-purpose core, because they cannot autonomously access operating system services like file systems or network stacks. Second, fast-path communication is at odds with preemptive context switching, which is still necessary today to improve efficiency when applications underutilize devices. Third, these concepts should be kept orthogonal, such that direct and unassisted communication is possible between any combination of accelerators and general-purpose cores. At the same time, all of them should support switching between multiple application contexts, which is most difficult with accelerators that lack the hardware features to run an operating system.

We present M³x, a system architecture that removes these roadblocks. M³x retains the low overhead of fast-path communication while enabling context switching for general-purpose cores and specialized accelerators. M³x runs accelerators autonomously and achieves a speedup of 4.7 for PCIe-attached image-processing accelerators compared to traditional assisted operation. At the same time, utilization of the host CPU is reduced by a factor of 30.

1 Introduction

The end of Dennard scaling [18] prevents further frequency gains and the prospect of dark silicon [21] hampers general-purpose parallelism. Hardware and system designers thus turn to new architectures to increase performance or reduce power consumption. These new ideas often revolve around specialization through custom accelerators [9, 26, 35, 37, 48, 67, 68] and streamlined communication that bypasses the operating system to avoid overheads [10, 39, 46].

TPUs [27] are a key example of the first approach. By creating a fixed-function accelerator for neural network training and inference, Google managed to increase performance per socket 30-fold and performance per watt 80-fold over a contemporary CPU. The second approach of preferring data fast paths to avoid indirections through the operating system can be observed today with technologies like single root I/O virtualization (SR-IOV) or Infiniband. System designs like M³ [10] and DLibOS [39] have shown that fast-path communication achieves latency reductions of 5× for a file system workload on M³, and 20× for memcached on DLibOS. Furthermore, our previous work M³ demonstrates that this idea can be generalized to provide fast-path communication between all compute units in the system.

Encouraged by these benefits, we expect ongoing development and increased deployment of these solutions. Use-case-driven accelerators will find their place in datacenters, also because of their deterministic execution model which helps to meet tail-latency requirements. We therefore assume that more applications will entail complex interactions between a mix of accelerators and general-purpose cores. Additionally, in multi-tenant cloud environments context switching is essential, because a single user will typically underutilize accelerators. We also envision advantages for small embedded and edge devices. Due to their limited hardware resources, these devices benefit from the power efficiency of accelerators and require context switching to flexibly time-share these resources.

1.1 Problem Statement

We extract three fundamental architectural challenges from our assumptions: First, the system architecture should enable accelerators to run autonomously. Currently, accelerators are often treated as peripheral devices whose execution needs to be assisted by a general-purpose CPU [57]. The TPUs described in Google's paper burden their controlling CPU with 11 – 76% load just to operate the TPU [27]. Our comparison to the traditional usage of accelerators in § 7.6 confirms this experience by showing that even a 3 GHz out-of-order

x86-64 core is 86% loaded to assist three image-processing accelerators attached via PCIe. If accelerators had direct access to data sources and sinks, this overhead would not be necessary. However, such connectedness requires first-party interaction of accelerators with OS services like storage and network. Specialized solutions exist, like GPUfs [58], GPUnet [29], and PTask [52] for GPUs or BORPH [59] and FPGAFS [31] for FPGAs. But there is no general solution that would grant any accelerator first-party access to OS services and also allow direct communication between multiple accelerators without assistance by a general-purpose core.

Second, fast-path communication without OS interaction is important for low-latency data and control transfer, but conflicts with context switching. This problem applies to communication channels involving general-purpose cores as well as accelerators. If communication partners are pinned and exclusively use dedicated resources, direct communication is easy. However, with context switching, communication needs to consider whether the intended recipient is currently running and how to deliver a message otherwise. A system design needs to answer, how the equivalent of a blocking system call should work on an accelerator that lacks the hardware features to run an operating system. Current solutions like M³ and DLibOS avoid this question and forgo context switching altogether.

Finally, all compute resources in the system — accelerators as well as general-purpose cores — should be accessible via the same communication primitives. The resulting system should enable developers to offload any job to the most suitable compute resource. Such unification was explored in previous work, but only for heterogeneous general-purpose cores [12, 19, 43].

1.2 Scope

We believe that accelerators should not be forced to adapt to operating system requirements, but focus on their main task: a fast and energy-efficient solution to a specific problem. In this paper, we take the extreme position and rethink the system architecture to enable a first-class integration of accelerators without imposing changes on them.

In contrast to conventional architectures, we do *not* build upon coherent shared memory for two reasons: First, providing global cache coherency is challenging for systems that consists of a wide variety of compute units such as general-purpose cores, DSPs, and fixed-function accelerators. Second, the costs of cache coherency in terms of chip area, power, complexity, and performance are expected to increase with an increasing number of compute units [28, 41]. For these reasons, it is still unclear whether and how future systems will support cache coherency. Therefore, we keep cache coherency optional.

Additionally, our long-term goal is to support arbitrary accelerators as first-class citizens. In this paper, we begin to address this challenge by demonstrating our approach for accelerators that are arguably the most difficult to support as first-class citizens: fixed-function accelerators [22, 26, 35, 37, 48, 65] that do not execute software and therefore provide none of the features that are required to run an operating-system kernel. We believe our efforts towards a unified interface for all compute units will generalize to more feature-rich accelerators in the future.

1.3 Contribution

We propose M³x, a solution for the identified issues using a hardware-software co-design approach.

- We explore the design space for fast-path communication and context switching. We explain the fundamental problems, when combining both techniques (§ 2) and discuss solutions in terms of interaction modes (autonomous vs. assisted) and mechanisms (hardware vs. software).
- We converge on a design for M³x that allows fast-path communication without involving the OS kernel and enables accelerators to access data sources and sinks without assistance by a general-purpose core. At the same time, M³x supports context switching on both general-purpose cores and accelerators (§ 4).
- We implement these mechanisms within the M³ OS and hardware architecture (§ 5). M³ already supports fast-path communication within a tile-based architecture and unifies communication among heterogeneous instruction sets [10]. Thus, it constitutes a suitable starting point, which we extend with support for context switching and autonomous accelerators.
- In the evaluation (§ 7), we demonstrate how M³x retains the low overhead of fast-path communication while enabling context switching. We show the performance and utilization benefits of autonomous accelerators using an accelerator benchmark suite and an application scenario that might occur in datacenters.

We rely on gem5 [16] as our simulation platform. Its high accuracy and modularity enable us to experiment with new hardware components. The implementation of M³x[1] and our extensions to gem5[2] are available as open source.

2 Background and Motivation

Traditional communication via UNIX pipes, sockets, or microkernel IPC involves the kernel in every communication. For that reason, the kernel can buffer messages until the recipient is ready to receive them, can schedule recipients based on pending messages, and can easily switch to a different thread if the current thread needs to wait for I/O. Communication that bypasses the kernel offers significant gains in terms of latency and throughput, as has been shown by M³ [10] and DLibOS [39]. We call such communication *fast-path communication* in this paper. Using fast-path

[1] https://github.com/TUD-OS/M3
[2] https://github.com/TUD-OS/gem5-dtu

communication with dedicated cores for the applications is easy, because none of the aforementioned actions are required, which is why M³ and DLibOS chose to omit context switching support altogether. We also believe the still increasing core counts and the dark silicon effect [21, 25] will reduce the context switching frequency and lead to dedicated cores for applications by default. However, in the foreseeable future, provisioning enough hardware resources to handle all load spikes is not feasible. These load spikes therefore require oversubscription of cores and accelerators. Thus, fast-path communication needs to be combined with context switching to use the system as efficiently as possible.

Combining fast-path communication with context switching is a hard problem, though. If the kernel is not involved in the communication, how can we determine whether the recipient is running and how can we deliver the message if it is not running? Even without relying on coherency (see § 1.2), we could buffer all messages in DRAM to cope with non-running recipients. However, this would effectively route all communication over DRAM, which increases latency and DRAM load and is therefore detrimental to the goals of fast-path communication. On the other hand, communicating directly between compute units, without involving the OS, has the consequence that a message cannot be delivered if the designated recipient is not running. The naive solution of waking up the recipient and retrying the fast-path communication is not sufficient. Since these two steps are not atomic, the recipient can be suspended in between, leading to no progress at the sender side. Furthermore, the kernel can no longer make scheduling or placement decisions if it cannot tell whether applications are currently waiting for a message or are doing useful work.

Accelerators typically lack the architectural features to run an OS kernel locally. To avoid the indirection of all communication through a remote kernel, accelerators require fast-path communication to interact with other accelerators [57]. However, as described before, fast-path communication is possible only if the recipient is running. Thus, the combination of fast-path communication and context switching is necessary to run accelerators autonomously, without indirection through the kernel.

3 Related Work

There are industry solutions for accelerator integration such as the coherent accelerator processor interface (CAPI) [6, 62] and the heterogeneous system architecture (HSA) [4, 51]. Both allow the integration of accelerators into a cache coherent virtual memory system, but in contrast to M³x, these hardware solutions do not consider direct access to operating system services by accelerators. Such access is investigated by other works for specific OS services and specific accelerators like GPUs [15, 29, 34, 52, 58, 64, 69] or FPGAs [31, 47, 59]. In contrast, M³x does not target a specific kind of accelerator, but provides a general construction principle for fast-path communication of any accelerator with any OS service or application.

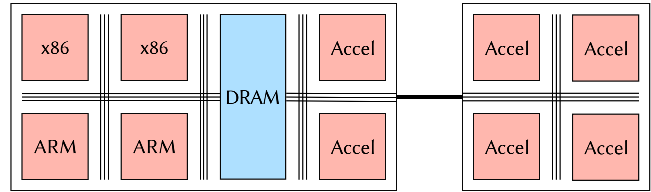


Figure 1: Overview of the system architecture.

K2 [36] and Popcorn Linux [12] demonstrate how the Linux kernel can be extended to support multiple coherence domains and potentially heterogeneous cores. However, heterogeneity in these cases is represented by general-purpose cores with different instruction sets and does not include fixed-function accelerators. Barrelfish [14] introduced the multikernel concept, where message passing is used to communicate between the operating system instances on each core. This concept is close to the design of M³x, but it assumes that all cores offer the architectural features to run an OS kernel. M³x sets out to remove this requirement to integrate fixed-function accelerators as well.

Arrakis [46] and OmniX [57] integrate peripherals and accelerators using SR-IOV. Instead of requiring the architectural features to run a kernel, these works assume the hardware to manage multiple contexts. M³x explores a more lightweight design, yet with enough hardware support to enable context switching and fast-path communication between components. Like NIX [11] or FlexSC [60], M³x adopts the idea of redirecting system calls to kernel cores to reduce the duties of non-kernel cores such as accelerators.

Horizontal system layouts [66] with different services on separate cores were explored in DLibOS [39] and M³ [10]. Both works have shown latency reductions due to the use of fast-path communication between cores, but they do not support context switching due to the problems explained in the previous section. Context switching of accelerators has been explored specifically for GPUs [13, 45], but without considering fast-path communication. M³x combines context-switching with the benefits of fast-path communication.

4 Design

We start this section with an introduction to the basic system architecture and discuss the design space for accelerator integration. Afterwards, we describe how fast-path communication is combined with context switching. Finally, we explain how this combination is used to run accelerators autonomously.

4.1 System Architecture

In this work, we assume a tiled system architecture as depicted in Fig. 1. The system uses an interconnect to communicate between tiles, similar to M³ [10] and DLibOS [39] and also similar to upcoming system architectures based on GenZ [3] or CCIX [2]. The tiles can contain heterogeneous *compute units* (CUs), ranging from general-purpose cores to DSPs

to fixed-function accelerators. These CUs can be part of the host system (left) and can be attached as an expansion card (right). The host system has a shared DRAM. We use the term *activity* to unify the active entities on these CUs. On general-purpose cores, an activity is typically a thread, whereas on accelerators it is the logic operating on a context.

4.2 Accelerator Integration

Adding accelerators into a system design poses the question of how to balance responsibilities between hardware and software. First, there are different ways to support arbitrary data sources and sinks for accelerators. Access to OS services like file systems or network stacks can be performed by software, which is the typical approach today. This approach requires a general-purpose core to assist the accelerator by continuously moving data back and forth. However, if the protocol to access OS services and data is sufficiently simple, it can be implemented in hardware. Such a hardware-friendly protocol allows accelerators to autonomously access arbitrary sources and sinks and removes software from the critical path. We present our protocol in § 4.8.

Second, if a system wants to support multiple activities with different priorities on a single accelerator, a low-latency context switch to the prioritized activity is needed. However, accelerators are typically invoked by software and are not interruptible until the computation is complete. One way to lower the latency is to reduce the amount of data per invocation. Consequently, the compute time per invocation is reduced, but software needs to continuously invoke the accelerator, which causes more CPU utilization and power consumption. Alternatively, the fine-grained invocation can be done in hardware by adding a simple state machine with preemption points next to the accelerator logic, as described in § 4.9.

Finally, to improve the utilization of accelerators, support for multiple contexts is necessary. One solution is to require the accelerator to provide a sufficient number of contexts and multiplex the hardware accordingly (e.g., based on SR-IOV). Alternatively, a combination of hardware and software can be used, which requires only a single context in hardware. To keep the hardware simple, we chose the latter approach: We perform the potentially complex scheduling decisions in software and add a simple state machine to the hardware, which saves and restores contexts.

4.3 Activity-aware Communication

To increase the flexibility and applicability of our system design, we chose not to rely on shared memory. Hence, messages cannot be delivered if the receiving activity is suspended (e.g., by preemption). There are two basic solutions to this problem:

1. *Eagerly* invalidate all incoming communication channels to an activity before suspending it or
2. keep the communication channels alive, but *lazily* detect communication attempts with suspended activities.

The eager approach does not require hardware support, but leads to more context switching overhead that grows linearly with the number of communication channels. In contrast, the lazy approach requires hardware support, but communication channels do not need to be invalidated on context switches. We chose the lazy approach, because our system supports many incoming communication channels (at most $(n-1)*m$ per compute unit with n CUs and m communication endpoints per CU). Furthermore, many communication channels are typically not used while an activity is suspended. To this end, we inform the hardware of the running activity and of the intended recipient activity when communicating (see § 5.5 for details). The hardware compares both and reports an error upon communication attempts with suspended activities.

4.4 Message Forwarding

Independent of eager invalidation or lazy detection, the hardware reports an error to the sender if the intended recipient is not running. Unfortunately, the naive solution of scheduling the recipient and retrying the fast-path communication introduces the following race condition: Since the kernel is not involved in this communication, it does not know when the communication has been completed successfully. If the kernel suspends the recipient before the communication has been finished, the sender does not make progress. The problem is that context switching and communication are decoupled, because the kernel performs the context switching, but activities bypass the kernel when performing fast-path communication. For example, if multiple senders try to communicate with multiple recipients scheduled on the same CU, the kernel could decide to schedule the next recipient before the communication with the current recipient has been finished.

We resolve this race condition by falling back to the traditional kernel-based communication model, if a communication failed due to a suspended activity. The kernel performs both the context switching and the communication: If activity A receives an error after trying to send a message to activity B, it asks the kernel to *forward* this message to B. When receiving the forward request, the kernel will first schedule B and afterwards send the message to B. To guarantee progress, the kernel does not suspend B until the message has been successfully delivered.

4.5 Computing vs. Idling

Another consequence of fast-path communication is that the kernel does not know whether an activity is currently computing or idling, because it waits for a message. We solve this problem by sending an *idle notification* to the kernel, similarly to scheduler activations [8]. Alternatively, the kernel could poll all CUs periodically to check whether the current activity is performing useful work, but we opted against this solution in favor of a less loaded and more scalable kernel.

We employ two optimizations. First, to prevent overeager context switches, we delay the sending of idle notifications

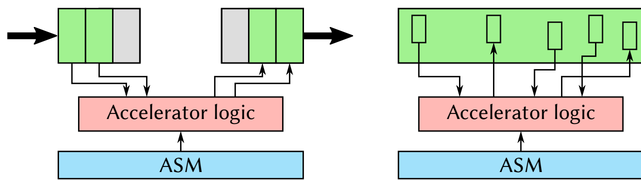


Figure 2: Stream-processing accelerators (left) and request-processing accelerators (right).

by a kernel-defined value called *idle delay*. The idle delay is stored in the address space of the current activity and updated by the kernel. Second, an activity does not need to send idle notifications at all if there is no ready activity that can run on its CU. In this case the idle delay is set to zero.

Note that we cannot force activities to report idling. However, threads on traditional systems can also decide to poll instead of using blocking system calls. On both systems, CPU-hogging activities can be penalized (e.g., priority degradation) and forcefully preempted.

4.6 Gang Scheduling

The concepts described so far allow to suspend activities, resume activities based on communication attempts, and to use the system's resources efficiently by switching to a different activity in case the current activity idles. However, if a set of heavily communicating activities contend with other activities for the same CUs, a systematic scheduling approach is required to maintain good performance. For example, consider a chain of accelerator activities that perform stream processing and therefore exchange messages and data at a high rate. If multiple such chains are contending for the same accelerators, the kernel needs to context switch these activities. However, uncoordinated context switching among the activities of all chains leads to many failed communication attempts when activities of different chains run simultaneously.

We solve this problem by introducing a simple form of gang scheduling [44]. Applications define the gang of a new activity at its creation time and the kernel pins all activities in a gang on different CUs and schedules them at the same time. We use this to run all activities of a single chain simultaneously. As the evaluation shows, multiple sets of heavily-communicating activities can therefore efficiently share the same CUs.

4.7 Accelerator Types

In this work, we consider two types of accelerators, depicted in Fig. 2: *stream-processing accelerators* that process a stream of data in blocks and compute on each block exactly once (e.g., AES encryption) and *request-processing accelerators* that receive the entire data for the computation with a single request and can access all data during the entire computation (e.g., graph processing or garbage collection). The stream-processing accelerators use DMA-based memory access to load a block of data from a source (e.g., a file or network socket), perform the computation on the block, and store the result to a sink. The request-processing accelerators use cache-based memory access to the request data to support large requests while maintaining fine-grained data access. For both accelerators, we add an *accelerator support module* (ASM), implemented as a finite-state machine, to the accelerator logic. The accelerator logic performs the computation, whereas the ASM interacts with other CUs and invokes the accelerator logic. We implemented the ASM as a piece of hardware in the current gem5-based prototype to demonstrate its simplicity.

Running these two types of accelerators autonomously requires access to OS services such as file systems, network stacks, and pipes to load and store data. Furthermore, accelerators need to be interruptible without requiring assistance by a general-purpose core. We describe our solution for both problems in the following sections.

4.8 Access to OS Services

Enabling accelerators to access files or network sockets requires a simple and unified protocol to obtain access to these resources. To this end, we designed a simple protocol for all file-like objects, in the same spirit as UNIX's everything-is-a-file principle. In contrast to UNIX, we implement OS services as microkernel-style servers and support both applications and accelerators as clients.

The *file protocol* uses a fast-path messaging channel between client and server. The server is expected to make the data available in memory and to provide the client with access to the data via a fast-path data channel. This channel enables accelerators to access large amounts of data autonomously, preventing frequent client-server interactions.

The protocol comprises two main requests: `next_in` and `next_out`. The former requests access to the next piece of data to read, whereas the latter requests access to the memory to which the next piece of data should be written. For example, a file-system server will provide the client with access to a fragmented file piece by piece, as described in more detail in § 5.7. After providing the client access to the data, the server returns the offset and size of the piece. Upon receiving this reply, the client can access the data via the fast-path data channel without involving the server again. After finishing the current piece, the client issues another `next_in` or `next_out` request to the server. A piece of length zero from the server denotes end-of-file.

As the client accesses the data on its own, the server does not know how many bytes the client has actually read or written. Therefore, input and output requests need to be *committed*. Each `next_in` and `next_out` request implicitly commits the complete previous piece of input or output data, respectively. Additionally, the `commit(nbytes)` request can be used to explicitly commit the first `nbytes` of the previous input or output request. The commit request is used, for example, if a client wants to stop writing to a file, in which case it might have written less than it got access to.

Finally, some servers support the seek request to change the file position. As described in more detail in § 5.8, the file protocol is implemented within the ASM of the stream-processing accelerators to load input data from arbitrary file-like sources and store the result to arbitrary file-like sinks. Note that request-processing accelerators can access OS services via the file protocol as well, but this has not been implemented. To test the generality of the protocol, we added a POSIX-like API on top and implemented a file system server, pipe server, and virtual terminal.

4.9 Interruptible Accelerators

As discussed in § 4.2, accelerators should be interruptible with low latency, requiring fine-grained invocations. At the same time, accelerators should run autonomously, asking for coarse-grained interactions with software. We achieve both by using the ASM as an indirection. Software performs the coarse-grained invocation of the hardware-implemented ASM, which in turn invokes the accelerator logic in a fine-grained fashion and is interruptible between these invocations. We implemented this scheme for request-processing accelerators, because the considered stream-processing accelerators already perform their computation block-wise with relatively small block sizes.

5 Implementation

Our prototype implementation is based on the hardware and software part of M³ [10]. The hardware platform of M³ exists by now as custom silicon in the Tomahawk 4 chip [24]. To extend the hardware part, we build on top of the already existing gem5 prototype. Both the gem5 prototype platform and the OS are open source and have been extended in this work to support context switching and autonomous accelerators.

5.1 Background on M³

The key idea of M³ is to introduce a new hardware component next to each CU, which serves as an abstraction for the heterogeneity of the CUs and supports fast-path communication between CUs. This hardware component is called data transfer unit (DTU) and is accessible over memory-mapped I/O (MMIO). Each CU is integrated with its DTU as a tile into the network-on-chip. The DTU provides a set of communication endpoints that can be configured as send, receive, or memory endpoints. Send and receive endpoints allow to establish a fast-path messaging channel, whereas memory endpoints are used for fast-path data channels. Data channels provide DMA-like access to a contiguous and byte-granular memory region.

The M³ kernel runs on a dedicated *kernel tile*, because not all tiles can be expected to run an OS kernel. The M³ kernel is different from traditional kernels, because it does not run user applications on the kernel tile. Instead, the kernel runs applications on other tiles, called *user tiles*, and waits for system calls in the form of messages, sent by applications via the DTU. Since only the kernel tile can configure DTU endpoints, applications are isolated from each other by default. The main responsibility of the kernel is to establish communication channels between applications by configuring DTU endpoints remotely. After a communication channel has been established, applications communicate directly with each other, bypassing the kernel.

On M³, the same activity abstraction[3] is used for all types of tiles, because the kernel is only concerned with their DTU state. The M³ kernel uses capabilities to manage the permissions in the system. Each activity has its own address space and capability space and system calls allow to exchange capabilities between activities. Since M³ does not support context switching, an activity is assigned to a free tile on creation and occupies this tile until its termination.

Outside of the kernel, M³ provides servers to host the actual functionality of the OS. M³ offers an in-memory filesystem, called *m3fs*, that organizes the data similarly to classical UNIX filesystems. The important difference is that m3fs grants applications direct access to file data via the DTU. Additionally, M³ offers a pipe server to connect activities via a unidirectional, first-in-first-out communication channel.

5.2 Virtual Memory Support

So far, M³ supports only simple general-purpose cores without virtual memory. Instead, cores have untranslated access to their dedicated scratchpad memory. To support more complex applications and be able to switch between them without first saving their entire memory state to DRAM, we added virtual memory support to M³. However, to prepare for future systems, M³x does not take advantage of cache coherency, but keeps it optional.

As virtual memory is also desirable for the cache-based memory access of request-processing accelerators, we added virtual memory support in two variants. For general-purpose cores, we use their memory management unit (MMU) and run a small helper on the core that receives page faults. For accelerators, we add an MMU to the DTU, consisting of a page table walker and translation lookaside buffer (TLB). In both cases, page faults are resolved by a *pager* in collaboration with the M³x kernel, which updates the page table entries, similar to other microkernel-based systems [33,61]. The pager is a server in M³x that supports copy-on-write and demand loading. On general-purpose cores, the helper sends a message to the pager to resolve page faults. On accelerators, the DTU sends the message to the pager, which is transparent to the accelerator.

5.3 Context Switching Overview

Context switches are performed remotely on the user tiles, initiated by the M³x kernel. This approach is required for accelerators that do not have the architectural features to run an OS kernel, but is optional for general-purpose cores with these features. In other words, the implementation could be extended to perform context switches on general-purpose cores locally.

[3] M³ calls tiles *processing elements* (PEs) and activities *virtual PEs* (VPEs).

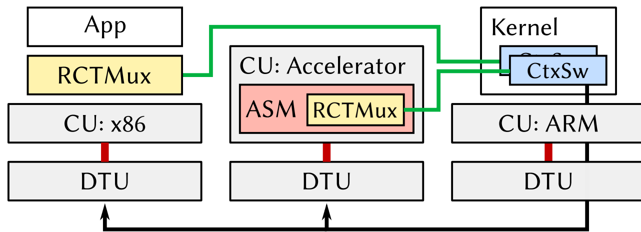


Figure 3: The involved components for context switches and their interfaces, shown on an exemplary assembly of CUs.

A context switch involves four components, depicted in Fig. 3: the CU, the DTU, the *context switcher* (CtxSw) in the M^3X kernel, and a small component on each user tile, called *remotely controlled time multiplexer* (RCTMux). RCTMux saves and restores the CU-state (e.g., CPU registers or the accelerator's local memory) during a context switch. The security-critical DTU-state (e.g., communication endpoints) is saved and restored by the kernel. RCTMux is CU-specific and either a piece of software on programmable CUs or a piece of hardware as part of the ASM for accelerators. The M^3X kernel maintains one context switcher for each user tile and performs scheduling and placement decisions.

These four components have two important interfaces. The first interface between the context switcher and RCTMux (green in Fig. 3) is used by the kernel to request saves and restores from RCTMux and by RCTMux to acknowledge their completion. Second, the DTU-CU interface (red) is used by the kernel to signal the CU about an imminent context switch. Depending on the type of CU, the signal injects an interrupt request into a core or notifies the ASM of an accelerator.

5.4 Kernel Extensions

We incorporated the context switcher module into the M^3 kernel to perform context switches on user tiles. First, the context switcher asks RCTMux to save the CU state. The context switcher then saves the DTU state of the current activity, restores the DTU state of the new activity, and asks RCTMux to restore said activity's CU state. Each of these steps is executed individually to be able to handle other requests (e.g., system calls) in the meantime.

Furthermore, we introduced a system call to forward messages upon communication attempts with suspended activities. The kernel buffers the message to forward, schedules the recipient, and delivers the message to the recipient as soon as it is running. Finally, we added a system call for idle notifications, upon which the kernel switches to the next ready activity or work-steals an activity from another tile in case no activity was ready. For application activities, the kernel sets the idle delay to 20,000 cycles[4]. For server activities, the kernel uses an idle delay of one cycle, because servers are typically only activated on demand. Hence, switching to an application is more beneficial for the system's performance.

To facilitate fast-path communication, the kernel migrates activities in two situations. First, if two activities are scheduled on the same CU and attempt to communicate, the kernel tries to migrate the currently suspended activity to another CU. If migration is not possible (e.g., no other compatible CU is available), the kernel instead performs a context switch from the activity that attempted the communication to the suspended activity. Second, if an activity is idling (see § 4.5), the kernel tries to work-steal a ready activity from a compatible CU.

5.5 DTU Extensions

To detect communication attempts with suspended activities, we equipped the DTU with the ID of the current activity and added the destination activity ID to the message header. If the destination activity ID at the recipient's DTU does not match the current activity ID, the DTU reports an error to the sender. In this case, the sender asks the kernel to forward the message to the recipient via the forward system call.

If the kernel decides to perform a context switch on a user tile, the DTU might currently be busy with a communication. As explained in § 5.1, the DTU supports messaging channels and data channels. Messages need to be delivered exactly once, whereas data accesses can be repeated. To keep the DTU simple, we decided against a complicated protocol to abort potentially ongoing communication. Instead, the DTU has an abort command, which consists of two parts. First, further communication attempts are rejected with an error until re-enabled by the kernel. Second, the DTU waits until all message-based communication is completed, whereas data accesses are aborted with an error and need to be repeated later.

5.6 RCTMux

We implemented RCTMux both for accelerators and for general-purpose cores. As mentioned before, on accelerators, RCTMux receives a signal from the kernel if a context switch is desired. The ASM checks for the signal only at convenient points in time, because the accelerator logic is not assumed to be interruptible. Upon the signal, it saves the ASM's state and the accelerator's local memory via DTU to a previously allocated space in DRAM, uses the DTU's abort command, and notifies the kernel that the state has been saved. Analogously, the state is restored upon a restore request from the kernel.

We also implemented RCTMux for x86-64 as a small piece of software running in ring 0. In this case, RCTMux is activated by an interrupt injection, saves the CPU registers, uses the DTU's abort command, and notifies the kernel. Upon a restore request from the kernel, it restores the CPU registers and resumes a previously aborted data access, if necessary.

5.7 File Protocol Servers

M^3 already features an in-memory file system, called m3fs, and a pipe server. However, since M^3 was only evaluated

[4]This idle delay turned out to be a good trade-off between context switching too often and overly long idle periods.

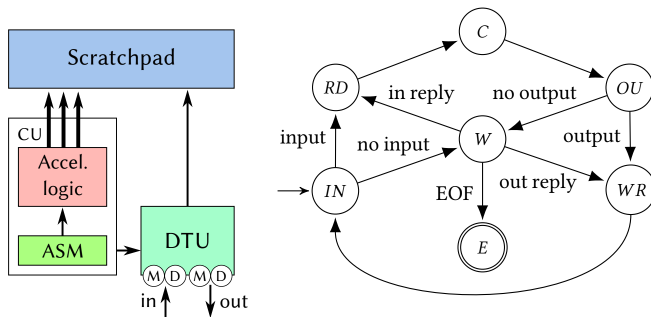


Figure 4: Stream-processing accelerator.

on general-purpose cores (in some cases with instruction extensions), the protocols to access these OS services are not suited for accelerators. First, M3 uses a different protocol for m3fs than for the pipe server, requiring accelerators to implement multiple protocols. Second, m3fs's protocol is based on the exchange of capabilities to obtain access to the data and requires clients to manage the file position. In summary, the existing protocols are too complex to be implemented in hardware. For that reason, we replaced them with the file protocol, as introduced in § 4.8.

On M3x, the file protocol is based on a messaging channel between client and server and a data channel to access the file data. Since m3fs manages the file data in extents, similar to other modern file systems [40, 50], the `next_in` request provides the client with access to the next extent of the file by asking the kernel to establish a corresponding data channel. The file position is managed and advanced by m3fs and can also be changed by the `seek` request. For appends, the `next_out` request allocates new space and provides the client with access to this space. Upon commit, m3fs truncates this space, if necessary, and makes it visible to other clients.

The pipe server uses a single and contiguous shared memory area in DRAM per pipe to exchange data between clients. For that reason, the pipe server asks the kernel to configure the client's data channel only once for the complete area and tells the clients where to read or write next. If no data can be read or written, the pipe server delays its response to the `next_in` or `next_out` request correspondingly.

5.8 File Protocol Clients

On the client side, we implemented the file protocol in software (for general-purpose cores) and in hardware (for accelerators). The software version is part of M3x's standard library and allows applications to use a POSIX-like file API. The library maps this API to the corresponding `next_in`, `next_out`, `commit`, and `seek` requests.

To enable access to file-like resources for stream-processing accelerators, we implemented the file protocol as part of the accelerator support module (ASM). The stream-processing accelerator has an input stream and an output stream, each using one messaging channel (M) to the server and one data channel (D) to access to data, as shown in Fig. 4. Like many other accelerators [17, 38, 55, 56, 63], the computation is performed on scratchpad memory (SPM), because it allows many parallel memory accesses (indicated by the thick arrows) and has predictable access latency.

The ASM loads data via the DTU from the input stream into the accelerator's SPM, activates the accelerator logic, and writes the result to the output stream. The ASM starts in state *IN*, which checks whether the input data channel has data left to read. If so, it directly transitions to state *RD* to read the next block of data into the SPM. Otherwise, it sends an input request (`next_in`) to the input server to request access to new input data and transitions to state *W*. State *W* waits until a message arrives and transitions to *RD* as soon as the reply to the input request has been received. After the next data block has been read into the SPM, the accelerator logic is activated and the ASM transitions to state *C* for the computation. As soon as the computation has been completed, the ASM transitions to state *OU*. Analogously to the input phase, *OU* first checks whether the output data channel has space left for the result of the computation. If so, it directly transitions to *WR* and writes the data. Otherwise it first requests new space from the output server (`next_out`). Afterwards, the ASM transitions back to state *IN*, which repeats the procedure until the reply to an input request indicates end-of-file. In this case, the ASM commits the written data by sending `commit` to the output server, if required, and transitions to state *E*.

6 Discussion

We believe that our architecture provides a good foundation for very heterogeneous systems, but we are aware that CUs will be diverse and have different requirements. This section discusses a few examples of how our current prototype can be extended to support other use cases.

Our context switching mechanism handles the simple save and restore actions on user tiles and the decision making in the M3x kernel. While we show in the evaluation that context switches on fixed-function accelerators have acceptable overhead, the mechanism is probably not a good fit for accelerators that have a large state such as GPUs. General-purpose cores provide native mechanisms to save/restore their state, which are used by the software version of RCTMux. Therefore we believe that large-state accelerators need tailored context-switching mechanisms as already partially supported by modern GPUs.

As described in § 5.4, the M3x kernel currently migrates an activity to a different tile if two activities on the same tile try to communicate. This policy assumes that the communication attempt starts a series of interactions between these activities, which mostly holds true for our current workloads. Clearly this is not the best solution in all cases. For example, if the activities communicate just once, a local communication channel with context switching can be preferable, if supported by the compute unit.

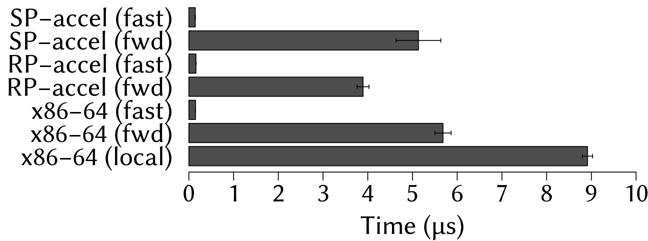


Figure 5: Fast-path vs. forwarded communication.

And finally, our current prototype does not queue messages at the recipient's compute unit if the recipient is suspended, but forwards the message to the recipient via the M³x kernel. We chose this solution to keep the hardware extensions small and, most importantly, minimize the burden on accelerators. If accelerators support message queues or such queues are added externally, the number of kernel involvements can be reduced. Thus, arguably our solution uses a queue size of zero, which can be extended to queue a few messages locally and only resort to the M³x kernel if the queue is full.

7 Evaluation

Our evaluation answers the following questions:

- How does fast-path and forwarded communication perform?
- Do the changes to the file protocol reduce its performance?
- What is the performance impact if activities share tiles?
- What are the benefits of autonomous accelerators?

7.1 Evaluation Platform

We used the gem5-based prototype platform for our evaluation. General-purpose tiles contain a single out-of-order x86-64 core with 32 KiB L1 instruction cache, 32 KiB L1 data cache, and 256 KiB L2 cache. The request-processing accelerators use 32 KiB L1 cache, whereas the stream-processing accelerators use 2 KiB scratchpad memory. General-purpose cores are simulated with a 3 GHz clock frequency, whereas accelerators are clocked with 1 GHz. All DTUs are configured to have 16 endpoints available. We use the `DDR3_1600_8x8` model of gem5 as the physical memory, clocked at 1 GHz. To keep the simulation times manageable, we connect the tiles via a crossbar instead of a full network-on-chip, which was sufficient, because our evaluation does not require large numbers of tiles. Due to the still long simulation times we used representative, but short-running benchmarks.

7.2 Fast-Path vs. Forwarding

M³x combines fast-path communication with context switching. In a first step, we use micro-benchmarks to determine the costs of forwarded communication, requiring a context switch, compared to fast-path communication. We measure the round-trip-time between activities on different CUs. Fig. 5 shows the average time over 16 runs with warm caches. The uppermost two rows show the time for stream-processing accelerators (SP), first if the recipient is running, resulting in fast-path communication, and second if the recipient is suspended, resulting in forwarded communication. The next two rows show the results for request-processing accelerators (RP), followed by two rows for an x86-64 core. The final row shows the time for a core-local round trip, using forwarded communication for both the request and the response.

As the results in Fig. 5 show, fast-path communication is more than one order of magnitude faster than forwarded communication on our system. All forwarded communication requires a forward system call, upon which the kernel performs a context switch to the receiving activity and forwards the message to the recipient. Most of the time is spent with the actual context switch, because it requires multiple steps and is carried out partially by RCTMux and partially by the kernel. Since stream-processing accelerators use a local scratchpad memory, the content needs to be saved and restored, leading to additional overhead. On x86-64, the overhead is larger, because the RCTMux is implemented in software. Finally, the core-local round trip requires two context switches. However, it is not twice as expensive as a single context switch, because the kernel optimizes this case by omitting the idle notification.

7.3 Application-level Benchmarks

As described in § 5.7, we simplified and unified the file protocol to be hardware-friendly. To evaluate whether these changes impact performance, we used the system call tracing infrastructure from M³. It allows to run an application on Linux, trace the system calls including timing information and replay the trace on M³. We used the following applications:

1. *tar*: creates a tar archive from files with sizes between 128 and 8192 KiB and 16 MiB in total,
2. *untar*: unpacks the same archive,
3. *shasum*: computes the SHA256 hash of a 512 KiB file,
4. *sort*: sorts a 256 KiB file with 408 lines,
5. *find*: searches 24 directories with 40 files each,
6. *SQLite*: creates a table and inserts/selects 32 entries, and
7. *LevelDB*: creates a table and inserts/selects 512 entries.

The applications *tar*, *untar*, *shasum*, *sort*, and *find* have been taken from BusyBox 1.26.2 [1]. *SQLite* is an embedded and highly reliable database engine [7]. *LevelDB* is a light-weight and high-performance key-value store, created by Google [5]. We chose these applications to stress the system in different ways: *tar* and *untar* are data intensive, *shasum* and *sort* are compute intensive, *find* performs many small file-system requests, and *SQLite* and *LevelDB* are mixtures of these.

We used these applications to compare the performance between Linux 4.10, M³ with the original file protocol, and M³x using the unified and hardware-friendly file protocol. In this section, M³ and M³x use three dedicated x86-64 tiles (without accelerator tiles) for the application, m3fs, and the pager, whereas Linux uses a single x86-64 core. However, M³ and M³x

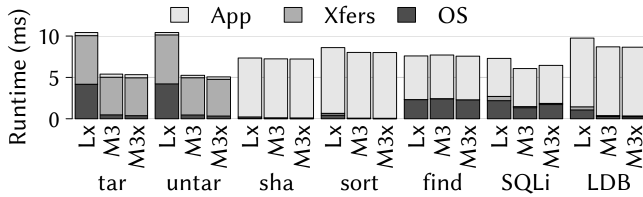


Figure 6: Performance comparison between Linux (Lx), M³, and M³x.

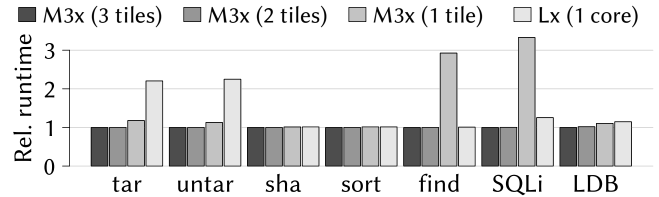


Figure 7: Application performance with a varying number of tiles, relative to the runtime on M³x with 3 tiles.

do not take advantage of multiple tiles, because all cross-tile interactions are synchronous and therefore, at no point in time is useful work done in parallel. On M³ and M³x, we use extents of at most 512 KiB, requiring multiple requests to m3fs to read and write files. On Linux, we use `tmpfs` as the in-memory file system. All file systems use a block size of 4 KiB. Fig. 6 shows the average runtime of 7 runs after one warmup run, broken down into the application time, time for data transfers, and OS overhead. We account Linux's time for the system calls, which are unsupported[5], as application time as well. Since the standard deviation is below 1%, we omit error bars.

In our previous work, we have already shown that data-intensive workloads like tar and untar have significantly less OS overhead on M³ than on Linux when running on simple Xtensa cores. As Fig. 6 shows, these improvements can be seen on x86-64 cores as well. Note however, that the difference is about a factor of two on x86-64 instead of five as on Xtensa, because the Xtensa cores did not have a cacheline prefetcher, resulting in poor performance on Linux [10]. On both architectures, the DTU's data channel can be configured in constant time for any byte-granular and contiguous region of memory, independent of its size. After the channel has been established, applications access the data via DMA with almost no overhead. Therefore, M³ and M³x outperform Linux significantly.

For the remaining applications, computation dominates the runtime, leading to smaller overall performance improvements. Note that SQLite is slightly faster on M³, because the new file protocol in M³x currently does not provide clients with read-write access to data and SQLite often switches between reading and writing of the same file. These switches require a `commit` request and a new `next_in` or `next_out` request, causing additional overhead.

7.4 Tile Sharing

After the performance comparison using three tiles, we show the performance impact when activities share tiles by means of context switching. In the first step, we ran both OS servers (m3fs and pager) on the same tile and in the second step, we ran the OS servers and the application on a single tile.

Fig. 7 shows the average runtime of three runs, preceded by one warmup run, normalized to the average runtime on M³x with three tiles. The standard deviation is less than 2%. As the results show, using the same tile for both servers and a dedicated tile for the application (two tiles in total) has almost no performance impact. Running servers and application on a single tile leads to a performance degradation in some cases. For *tar* and *untar*, the runtime is increased by 17% and 12%, respectively, but M³x is still about twice as fast as Linux. *shasum* and *sort* show almost no performance degradation, whereas *find* and *SQLite* experience a significant slowdown. The reason is, that both *find* and *SQLite* communicate heavily with m3fs, leading to many context switches. The performance of *LevelDB* degrades slightly, but is still better than on Linux. We conclude that some workloads require faster core-local context switches. We could improve M³x by running a kernel with context-switching support directly on the core, in case the necessary hardware features are available.

7.5 Autonomous Request Processing

After the evaluation on general-purpose cores, we want to demonstrate the benefits of autonomous accelerators. In this section, we start with request-processing accelerators. As described in § 4.9, software invokes the ASM, which in turn invokes the accelerator logic. In this section, we evaluate the impact of the invocation granularity on performance and CPU wake-up frequency.

We simulate the accelerators using gem5-Aladdin [56], which is a power-performance accelerator modeling framework that can be used to explore the design space for fixed-function accelerators. gem5-Aladdin simulates the accelerator logic and uses the memory subsystem of gem5 to perform memory accesses. gem5-Aladdin achieves an error of less than 6% for the accelerator's performance compared to real hardware. We adapted gem5-Aladdin to be invoked by the ASM and to notify the ASM of completions.

To use a request-processing accelerator, an application creates an activity for the desired accelerator, creates the memory mappings for the input and output data in the activity's virtual address space, and establishes the communication channel to invoke the ASM. In this case, the input data is stored in files and the output data should be written to files as well. Therefore, these files are mapped into the virtual address space of the accelerator activity.

We use different accelerator workloads from MachSuite [49]. MachSuite has been analyzed by gem5-Aladdin

[5] In these benchmarks, the system calls `access`, `brk`, `chdir`, `chmod`, `chown`, `dup2`, `fchown`, `fcntl`, `fdatasync`, `futex`, `geteuid`, `getpid`, `getrlimit`, `gettimeofday`, `getuid`, `ioctl`, and `utimes` were unsupported on M³/M³x. The sum of the times for the ignored system calls was at most 0.4 ms.

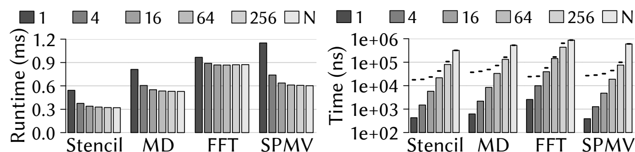


Figure 8: Total runtime (left) and the average (bars on the right) and maximum execution times (lines on the right) for different batch sizes.

with the result that some accelerators benefit from DMA-based memory access and others benefit from cache-based memory access. For this evaluation, we picked the accelerators that benefit from cache-based memory access:

1. Stencil-3D: a three-dimensional stencil computation,
2. MD-KNN: a k-nearest-neighbor computation from molecular dynamics,
3. FFT-1D: a one-dimensional fast Fourier transform, and
4. SPMV: a sparse matrix-vector multiplication.

We adjusted each accelerator to perform a single indivisible step per invocation by the ASM. Multiple such invocations are batched in a single invocation by the CPU. We analyze the spectrum between assisted and autonomous operation by varying the batch size.

Fig. 8 shows the results from three runs after one warmup run with batch sizes of 1 to 256. Performing all invocations in a single batch is shown as 'N', because the total number of invocations depends on the workload. The standard deviation is less than 1%. As can be seen in the left part of the figure, larger batch sizes lead to better performance. More importantly, the right part of the figure shows the average (bars) and maximum (lines) accelerator execution times for each ASM invocation. For example, using the MD workload and a batch size of 16 shows acceptable performance, but leads to a context switching latency of 48 µs and the CPU is woken up every 8 µs on average. High wake-up frequencies are a problem on modern cores, which can only achieve significant power savings in deep sleep states. However, the deeper the sleep state, the longer the time to bring the core back into a functional state (e.g., on Intel's Haswell generation, dozens of microseconds to leave C6 and up to several milliseconds to leave C10 [32, 54]). Hence, deeper sleep states are only beneficial during longer idle periods.

M³X performs all accelerator invocations in a single batch and uses an ASM that is interruptible between invocations to get the best of both worlds: On the one hand, a single batch leads to the best performance. On the other hand, the fine-grained interruptibility allows to context-switch to a more important activity with a low latency. Note that an immediate interruption can be achieved by resetting the accelerator logic, but requires to repeat the last step of the computation. If all invocations are done by the ASM in hardware (*autonomous*), the accelerator needs to repeat only a single indivisible step. If all invocations are done in software (*assisted*), the accelerator needs to repeat as many steps as the performance and energy constraints allow.

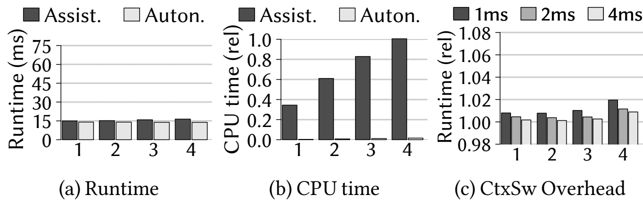
7.6 Autonomous Stream Processing

Finally, we want to show the benefits of autonomous stream-processing accelerators. Stream processing is used in various domains such as mobile communication, image processing, and audio processing. In this work, we consider an image processing scenario as is imaginable in data centers, similar to Google's TPU [27] workloads. The cloud provider offers a set of image-processing accelerators as a service and allows customers to perform large-scale image processing on these accelerators. An efficient method for large images is FFT convolution [42], which first performs a 2D fast Fourier transform (FFT) on the input image, then multiplies the result pointwise with an image filter, and finally performs the inverse FFT. Depending on the filter, FFT convolution can be used, for example, for edge detection or low-pass filtering.

To evaluate this scenario, we use three types of accelerators called FFT, MUL, and IFFT. Each accelerator has 2 KiB (the block size for the 32×32 point FFT) of local scratchpad memory (SPM) and uses the file protocol (see § 5.8) to stream the data block-wise from the input stream via the SPM to the output stream. Due to the deterministic execution model of these accelerators, we did not use gem5-Aladdin to simulate the accelerator logic, but used Aladdin [55] to determine the computation times offline. To get reasonable results, we generated all sensible configurations and picked the sweet spot between performance and the product of chip area and power consumption. We obtained 5,856 cycles for FFT and IFFT and 1,189 cycles for MUL. To put these numbers into perspective, the FFT/IFFT accelerator is about three times as fast as a simple software implementation and Aladdin reports a three orders of magnitude lower power consumption than a typical modern x86 core.

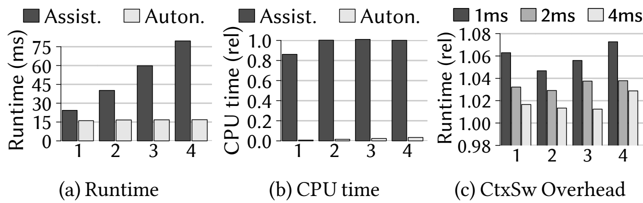
These three types of accelerators run activities that form an FFT-MUL-IFFT chain to process a 4 MiB image file and store the resulting image as a file. In our first experiment, we run 1 to 4 such chains simultaneously without context switching, thus using 1 to 4 instances of each accelerator type. To show the benefits of autonomous accelerators, we compare M³X's autonomous approach with the assisted approach. The assisted approach drives the accelerators from software using a single general-purpose core. Hence, software loads the input data into the SPM, starts the accelerator via a message, and asks the accelerator's DTU to move the result from the SPM to the next accelerator or to the output file. The autonomous variant connects the DTU endpoints of the accelerators as follows: The input of the first and the output of the last accelerator are connected to a file. The output of the first and second accelerator are directly pushed to the successor.

We simulate two ways to attach accelerators to the system: network-on-chip and PCI Express (PCIe). The former leads to superior performance due to lower latency, whereas the latter



(a) Runtime (b) CPU time (c) CtxSw Overhead

Figure 9: Total runtime, CPU time, and context switching overhead for different numbers of accelerator chains when integrating the accelerators into the NoC.



(a) Runtime (b) CPU time (c) CtxSw Overhead

Figure 10: Like Fig. 9, but with PCIe-attached accelerators.

allows a flexible combination of independently developed components. The result for the NoC version is depicted in Fig. 9, whereas the PCIe version is depicted in Fig. 10. We show the overall runtime (a) and the CPU time spent to drive the accelerators (b), depending on the number of accelerator chains using three runs, preceded by one warmup run. The standard deviation is below 3%. We simulate the PCIe-attached add-on card by connecting the accelerators via a bridge with a delay of 500 ns to the host system, which is the typical one-way latency for PCIe gen 3 [20, 23, 30, 53]. In other words, we do not simulate a complete PCIe interconnect, but only the latency PCIe introduces. The one-way latency within the NoC is about 10 ns. In both cases, the DRAM is part of the host system and stores the in-memory file system.

Using the assisted approach leads to slightly worse overall runtime with an increasing number of accelerator chains when integrating the accelerators into the NoC. With PCIe, the overall runtime increases significantly, leading to a slowdown of factor 4.7. In contrast, the autonomous approach always achieves the same runtime, independent of the number of chains. More importantly, the assisted approach keeps the CPU busy most of the time. Within the NoC, the CPU is utilized 100% of the time starting at four accelerator chains, whereas with PCIe, the CPU is already fully utilized starting with two chains. The autonomous approach does not cause significant CPU load in either case. Additionally, Fig. 10 shows that the autonomous approach outperforms the assisted approach for PCIe-based accelerators even if the assisted approach does not fully utilize the CPU. The reason is the 500 ns delay when communicating with the accelerators, which prevents the assisted approach from fully utilizing the accelerators.

Finally, we evaluate the context switching overhead when two chains of activities compete for the same accelerators. In this case, we use only the autonomous approach and put each chain of activities into the same gang to benefit from gang scheduling. Plot (c) in Fig. 9 and Fig. 10 shows the context switching overhead by comparing the runtime of two activity chains running consecutively with the runtime of two chains running interleaved. We vary the time slice length for context switching between 1 ms and 4 ms. As the results show, using a still rather short time slice of 4 ms leads to less than 0.9% overhead with accelerators integrated into the NoC and less than 2.9% overhead when attaching them via PCIe.

Note that the performance can still be improved for both the assisted and the autonomous approach. For the assisted approach, batching could be used to reduce the interaction frequency with the accelerators. However, batching is only possible by increasing the SPM sizes of the accelerators, which is expensive in terms of area and energy and increases the time the accelerator is not interruptible. Additionally, the assisted approach can trade more CPU time for more accelerator performance by using multiple cores to drive the accelerators, until the PCIe bus becomes the bottleneck. The autonomous approach does not suffer from the trade-off between SPM size and CPU utilization and can further improve performance by overlapping data transfers to the DRAM instead of issuing one transfer at a time.

8 Conclusion

In this work, we presented M³x, which combines fast-path communication, bypassing the kernel, with context switching and thereby enables autonomous accelerators. To this end, we re-evaluated the boundary between hardware and software. We found that (1) introducing a hardware-friendly file protocol enables accelerators to autonomously access file systems or network stacks, (2) performing potentially complex scheduling decisions in software and simple save and restore actions in hardware allows to context switch accelerators, and (3) attaching a simple hardware component to the accelerator logic allows to combine autonomous operation and interruptibility.

We demonstrated in our evaluation that M³x retains the performance advantages of M³'s fast-path communication, while using the system's resources more efficiently by performing context switches, if required. Additionally, we have shown that running PCIe-attached image processing accelerators autonomously achieves a speedup of 4.7 and reduces the CPU utilization by a factor of 30.

In future work, we plan to study other types of accelerators such as FPGAs and GPUs and apply our insights from simulation to real hardware.

9 Acknowledgments

We would like to thank our shepherd, Christopher Rossbach, and the anonymous reviewers for their helpful suggestions. This work was funded by the German Research Council DFG through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed) and by public funding of the state of Saxony/Germany.

References

[1] BusyBox. https://www.busybox.net. Accessed: 10/27/2018.

[2] Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com. Accessed: 08/03/2018.

[3] Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access. https://genzconsortium.org/. Accessed: 08/03/2018.

[4] HSA foundation ARM, AMD, Imagination, MediaTek, Qualcomm, Samsung, TI. http://www.hsafoundation.com. Accessed: 12/15/2017.

[5] LevelDB. https://leveldb.org. Accessed: 06/15/2018.

[6] OpenCAPI consortium. https://opencapi.org/. Accessed: 12/15/2017.

[7] SQLite. https://www.sqlite.org. Accessed: 07/12/2017.

[8] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[9] Arnold, O., Matus, E., Noethen, B., Winter, M., Limberg, T., and Fettweis, G. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS) 13*, 3s (Mar 2014), 107:1–107:24.

[10] Asmussen, N., Völp, M., Nöthen, B., Härtig, H., and Fettweis, G. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS'16, ACM, pp. 189–203.

[11] Ballesteros, F. J., Evans, N., Forsyth, C., Guardiola, G., McKie, J., Minnich, R., and Soriano-Salvador, E. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal 17*, 2 (2012), 41–54.

[12] Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., and Ravindran, B. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys'15, ACM, pp. 29:1–29:16.

[13] Basaran, C., and Kang, K.-D. Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems* (Washington, DC, USA, 2012), ECRTS'12, IEEE Computer Society, pp. 287–296.

[14] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 29–44.

[15] Bergman, S., Brokhman, T., Cohen, T., and Silberstein, M. SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the Seventeenth USENIX Annual Technical Conference* (2017), vol. 17 of *USENIX ATC'17*.

[16] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Computer Architecture News 39*, 2 (u 2011), 1–7.

[17] Cota, E. G., Mantovani, P., Di Guglielmo, G., and Carloni, L. P. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Annual Design Automation Conference* (New York, NY, USA, 2015), DAC'15, ACM, pp. 202:1–202:6.

[18] Dennard, R., Rideout, V., Bassous, E., and LeBlanc, A. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (Oct 1974), 256–268.

[19] DeVuyst, M., Venkat, A., and Tullsen, D. M. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS'12, ACM, pp. 261–272.

[20] Erickson, K. G., Boyer, M. D., and Higgins, D. NSTX-U advances in real-time deterministic PCIe-based internode communication. *Fusion Engineering and Design 133* (2018), 104–109.

[21] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA'11, ACM, pp. 365–376.

[22] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. *IEEE Micro 33*, 3 (May 2013), 16–27.

[23] Flajslik, M., and Rosenblum, M. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC'13, USENIX, pp. 333–346.

[24] Haas, S., Seifert, T., Nöthen, B., Scholze, S., Höppner, S., Dixius, A., Adeva, E. P., Augustin, T., Pauls, F., Moriam, S., Hasler, M., Fischer, E., Chen, Y., Matúš, E., Ellguth, G., Hartmann, S., Schiefer, S., Cederström, L., Walter, D., Henker, S., Hänzsche, S., Uhlig, J., Eisenreich, H., Weithoffer, S., Wehn, N., Schüffny, R., Mayr, C., and Fettweis, G. A heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications. In *Proceedings of the 54th Annual Design Automation Conference 2017* (New York, NY, USA, 2017), DAC'17, ACM, pp. 47:1–47:6.

[25] Henkel, J., Khdr, H., Pagani, S., and Shafique, M. New trends in dark silicon. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference* (2015), DAC'15, IEEE, pp. 1–6.

[26] Jarvinen, K., and Skytta, J. High-speed elliptic curve cryptography accelerator for koblitz curves. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines* (April 2008), FCCM'08, pp. 109–118.

[27] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA'17, ACM, pp. 1–12.

[28] Kelm, J. H., Johnson, D. R., Tuohy, W., Lumetta, S. S., and Patel, S. J. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE micro 31*, 1 (2011), 42–55.

[29] Kim, S., Huh, S., Hu, Y., Zhang, X., Witchel, E., Wated, A., and Silberstein, M. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 201–216.

[30] Kim, S., and Yang, J.-S. Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system. In *Proceedings of the 55th Annual Design Automation Conference* (2018), DAC'18, ACM, p. 56.

[31] Krill, B., Amira, A., and Rabah, H. Generic virtual filesystems for reconfigurable devices. In *2012 IEEE International Symposium on Circuits and Systems* (2012), IEEE, pp. 1815–1818.

[32] Kurd, N., Chowdhury, M., Burton, E., Thomas, T. P., Mozak, C., Boswell, B., Mosalikanti, P., Neidengard, M., Deval, A., Khanna, A., et al. Haswell: A family of IA 22 nm processors. *IEEE Journal of Solid-State Circuits 50*, 1 (2015), 49–58.

[33] Lackorzynski, A., and Warg, A. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (New York, NY, USA, 2009), IIES'09, ACM, pp. 25–30.

[34] LeBeane, M., Potter, B., Pan, A., Dutu, A., Agarwala, V., Lee, W., Majeti, D., Ghimire, B., Van Tassell, E., Wasmundt, S., Benton, B., Breternitz, M., Chu, M. L., Thottethodi, M., John, L. K., and Reinhardt, S. K. Extended task queuing: Active messages for heterogeneous systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2016), SC '16, IEEE Press, pp. 80:1–80:12.

[35] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA'13, ACM, pp. 36–47.

[36] Lin, F. X., Wang, Z., and Zhong, L. K2: A mobile operating system for heterogeneous coherence domains. *ACM Transactions on Computer Systems 33*, 2 (u 2015), 4:1–4:27.

[37] Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS'15, ACM, pp. 369–381.

[38] Liu, Z., Severance, A., Singh, S., and Lemieux, G. G. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays* (2012), FPGA'12, ACM, pp. 229–232.

[39] Mallon, S., Gramoli, V., and Jourjon, G. DLibOS: Performance and protection with a network-on-chip. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS'18, ACM, pp. 737–750.

[40] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 21–33.

[41] Mattson, T. G., Van der Wijngaart, R., and Frumkin, M. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC'08, IEEE Press, pp. 38:1–38:11.

[42] Moreland, K., and Angel, E. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 112–119.

[43] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 221–234.

[44] Ousterhout, J. K., et al. Scheduling techniques for concurrent systems. In *ICDCS* (1982), vol. 82, pp. 22–30.

[45] Park, J. J. K., Park, Y., and Mahlke, S. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS'15, ACM, pp. 593–606.

[46] Peter, S., Li, J., Zhang, I., Ports, D. R., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems 33*, 4 (2016), 11.

[47] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 13–24.

[48] Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C., and Horowitz, M. A. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 24–35.

[49] Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y., and Brooks, D. Machsuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2014), IISWC'14, IEEE, pp. 110–119.

[50] Rodeh, O., Bacik, J., and Mason, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug 2013), 9:1–9:32.

[51] Rogers, P., and Fellow, A. Heterogeneous system architecture overview. In *Hot Chips* (2013), vol. 25.

[52] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP'11, ACM, pp. 233–248.

[53] Rota, L., Vogelgesang, M., Perez, L. A., Caselle, M., Chilingaryan, S., Dritschler, T., Zilio, N., Kopmann, A., Balzer, M., and Weber, M. A high-throughput readout architecture based on PCI-Express Gen3 and DirectGMA technology. *Journal of Instrumentation 11*, 02 (2016), P02007.

[54] Schöne, R., Molka, D., and Werner, M. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development 30*, 2 (2015), 219–227.

[55] Shao, Y. S., Reagen, B., Wei, G.-Y., and Brooks, D. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st Annual International Symposium on Computer Architecture* (2014), ISCA'14, IEEE, pp. 97–108.

[56] Shao, Y. S., Xi, S. L., Srinivasan, V., Wei, G.-Y., and Brooks, D. Co-designing accelerators and SoC interfaces using gem5-aladdin. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016), MICRO'16, IEEE, pp. 1–12.

[57] Silberstein, M. OmniX: An accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS'17, ACM, pp. 69–75.

[58] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS'13, ACM, pp. 485–498.

[59] So, H. K.-H., and Brodersen, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems 7*, 2 (Jan 2008), 14:1–14:28.

[60] Soares, L., and Stumm, M. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[61] Steinberg, U., and Kauer, B. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys'10, ACM, pp. 209–222.

[62] Stuecheli, J., Blaner, B., Johns, C., and Siegel, M. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development 59*, 1 (2015), 7–1.

[63] Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. A. Performance and energy limits of a processor-integrated FFT accelerator. In *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference* (2014), HPEC'14, IEEE, pp. 1–6.

[64] Tseng, H.-W., Zhao, Q., Zhou, Y., Gahagan, M., and Swanson, S. Morpheus: creating application objects efficiently for heterogeneous computing. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture* (2016), ISCA'16, IEEE, pp. 53–65.

[65] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 205–218.

[66] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43*, 2 (Apr 2009), 76–85.

[67] Wu, L., Barker, R. J., Kim, M. A., and Ross, K. A. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA'13, ACM, pp. 249–260.

[68] Yu, W., and He, Y. A high performance CABAC decoding architecture. *IEEE Transactions on Consumer Electronics 51*, 4 (Nov 2005), 1352–1359.

[69] Zhang, J., Donofrio, D., Shalf, J., Kandemir, M. T., and Jung, M. NVMMU: A non-volatile memory management unit for heterogeneous GPU-SSD architectures. In *Proceedings of the International Conference on Parallel Architecture and Compilation* (2015), PACT'15, IEEE, pp. 13–24.

SmartDedup: Optimizing Deduplication for Resource-constrained Devices

Qirui Yang
Arizona State University

Runyu Jin
Arizona State University

Ming Zhao
Arizona State University

Abstract

Storage on smart devices such as smartphones and the Internet of Things has limited performance, capacity, and endurance. Deduplication has the potential to address these limitations by eliminating redundant I/Os and data, but it must be considered under the various resource constraints of the devices. This paper presents SmartDedup, a deduplication solution optimized for resource-constrained devices. It proposes a novel architecture that supports symbiotic in-line and out-of-line deduplication to take advantage of their complementary strengths and allow them to be adapted according to a device's current resource availability. It also cohesively combines in-memory and on-disk fingerprint stores to minimize the memory overhead while achieving a good level of deduplication. SmartDedup is prototyped on EXT4 and F2FS and evaluated using benchmarks, workloads generated from real-world device images, and traces collected from real-world devices. The results show that SmartDedup substantially improves I/O performance (e.g., increases write and read throughput by 31.1% and 32%, respectively for an FIO experiment with 25% duplication ratio), reduces flash writes (e.g., by 70.9% in a trace replay experiment with 75.8% duplication ratio), and saves space usage (e.g., by 45% in a DEDISbench experiment with 46.1% duplication ratio) with low memory, storage, and battery overhead, compared to both native file systems and related deduplication solutions.

1 Introduction

Smart devices such as smartphones and the Internet of Things (IoT) are becoming pervasively used. The high volume and velocity of data produced by the growing applications and sensors on these devices present serious challenges to the on-device flash storage, which has limited performance, capacity, and endurance. Deduplication has the potential to address these limitations by reducing the I/Os and storage caused by duplicate data. But adopting deduplication on smart devices must address their unique resource constraints. In particular, the *limited memory* on the devices presents a difficult trade-off between the speed of deduplication and the amount of duplicates that it can find. The *limited storage performance and endurance* require the deduplication operations to incur minimal additional I/Os. For many devices with *limited power and energy capacity*, the design of deduplication also needs to be aware of its power and energy usage.

To address these challenges, this paper presents SmartDedup, a deduplication solution optimized for smart devices considering their various resource constraints. The architecture of SmartDedup is designed to support the symbiotic in-line and out-of-line deduplication. It employs in-line deduplication in the foreground to reduce redundant writes and runs out-of-line deduplication in the background to find duplicates missed by in-line deduplication. These two techniques work together cooperatively by sharing the fingerprint stores and information (e.g., fingerprints) about the I/O requests.

SmartDedup employs cohesively designed in-memory and on-disk fingerprint stores to minimize memory overhead while achieving a good level of deduplication. The small in-memory fingerprint store provides fast fingerprint lookup and effective write reduction; the on-disk fingerprint store supports complete data deduplication. Fingerprints migrate between the two stores dynamically based on data access patterns. SmartDedup also embodies several techniques to make efficient use of the two fingerprint stores. To further reduce the overhead, both fingerprint stores share the same indexing data structure to save memory usage; fingerprints are evicted from memory in groups to reduce the involved I/Os and wear-

out.

To support this study, we collected file system images and long-term (2-6 months) traces from real-world smartphones. The data confirms that there is a good level of duplication in real-world workloads: the average duplication ratio is 33% in the data from the images, and ranges from 22% to 48% among the writes in the traces. The specific applications and types of files that contribute the most duplicates differ by device, so a holistic system-level deduplication solution is needed to fully exploit these I/O and data reduction opportunities.

We prototyped SmartDedup on OSs (Android and Raspbian) and file systems (EXT4 and F2FS) commonly used by smart devices. We evaluated it on two representative devices, a Nexus 5X smartphone and a Raspberry Pi 3 device, using intensive benchmark (FIO [12]) and realistic workloads generated by DEDISbench [20] from sampling real-world smartphone images and by replaying real-world smartphone I/O traces. The results show that SmartDedup achieves substantial improvements in performance, storage utilization, and flash endurance compared to both the native file systems and related solutions.

For example, on Nexus, SmartDedup outperforms native EXT4 by 16.9% in throughput and 18.5% in 95th percentile latency for writes, and 38.2% in throughput and 18.7% in 95th percentile latency for reads, for intensive FIO workloads with only 25% duplicates; and it improves the write latency by 25.5% for a trace segment with a 47.9% duplication ratio. In terms of space saving, SmartDedup saves 45% of space (after factoring in its own space overhead) compared to EXT4 when the duplication ratio is 46.1% in a DEDISbench experiment. In terms of reducing the wear-out, SmartDedup reduces 70.9% of writes from EXT4 (after factoring in its own write overhead) in a trace replay with a duplication ratio of 75.8%. SmartDedup also outperforms the state-of-the-art related works Dmdedup [16, 23] by 1.5X in throughput and 57.1% in 95th percentile latency in an FIO write experiment with 25% duplicates, and CAFTL [9] by 37.7% in average write latency in a trace replay with 75.8% duplication ratio. All the improvements are achieved with low resource usage. In these experiments, SmartDedup uses less than 3.5MB of memory, and it actually reduces the battery usage by up to 7.7% and 49.2% for intensive workloads on Nexus and Pi, respectively.

2 Analysis of Real-world Device Data

To confirm the potential of deduplication for smart devices, we collected and analyzed I/O traces and file system images from smartphones used by different real users.

ID	Total I/Os (GB)	Read/ write ratio	# of days	Unique addresses (GB)	Unique data (GB)	Duplication ratio (%)
1	2096.8	7.5	173	148.2	192.7	21.9
2	773.6	3.4	85	36.3	96.2	45.3
3	426.7	4.8	73	34.6	56.5	23.2
4	2385.6	5.8	145	101.2	184.2	47.5
5	1119.7	3.4	92	126.7	148.6	41.6
6	765.9	3.4	72	23.9	124.8	28.3

Table 1: File system traces from real-world smartphones. For each trace, the table shows the total amount of data requests (4KB each), the ratio between reads and writes, the total number of days, the total amount of writes with unique addresses, the total amount of writes with unique data, and the total percentage of duplicate writes.

	Top 3 duplicate contributors (file type, % of writes, duplication ratio (%))		
Trace	1	2	3
1	(res, 12, 15)	(db, 10, 15)	(exe, 39, 13)
2	(tmp, 3, 73)	(exe, 36, 65)	(res, 27, 62)
3	(res, 17, 65)	(media, 3, 19)	(db, 11, 17)
4	(exe, 29, 78)	(tmp, 4, 77)	(media, 24, 49)
5	(media, 76, 48)	(res, 2, 34)	(exe, 12, 25)
6	(res, 19, 45)	(exe, 35, 36)	(media, 9, 31)

Table 2: The file types that contribute the most amount of duplicates in the file system traces collected from smartphones used by real users. For each top contributor, the table shows the file type, the percentage of writes that it contributes to the trace's total write volume, and the percentage of duplicate writes within only this type of files.

First, we studied the long-term file system I/O traces [4] collected from six smartphones (from VFS and EXT4 on Android) used by six users from four countries who are between 20 to 40 years old, which recorded the fingerprints of writes when they were flushed from the page cache to flash storage. Commonly used applications include chat (WhatsApp and WeChat), video (Youtube), and social network (Facebook and Weibo) applications. As summarized in Table 1, these traces confirm that: 1) real-world device workloads are indeed quite intensive, and do have a significant impact on the performance and endurance of devices. The average daily I/O volume ranges from 4.2GB to 17.6GB, and the amount of writes ranges from 0.7GB to 3.5GB; and 2) a good level of deduplication can be achieved on the writes captured in the traces. Considering the entire traces, the percentage of duplicate writes ranges from 21.9% to 47.5%; and on a daily basis, on average between 16.6% and 37.4% of writes are also duplicates.

To understand where the duplicates came from, we further analyzed the effectiveness of deduplication within

the writes to each type of files. We classified the files into several categories (*resource* files, *database* files, *executables*, *temporary* files, and *multimedia*), following the methodology in [14]. [1] Table 2 shows that the file types that contribute the most duplicates vary across the traces collected from different users' devices. This observation suggests that applying deduplication only to specific types of files, or even more narrowly, only to specific applications [17], is insufficient. Although not shown in the figure, our results also reveal that whole-file-based deduplication [3] is also insufficient as over 80% of the duplicates are from files that are not completely duplicate.

To complement the above trace analysis, we also studied file system images collected from 19 real-world smartphones, which on average have 10.4GB of data stored on the device and 33% duplicates in the data. The analysis also confirms that there is a good amount of duplicate data stored on the devices. We also analyzed the effectiveness of deduplication on different types of files, and as in the trace analysis, we did not find any pattern—the file types that contribute the most to deduplication differ across the devices. For example, on one image, a large percentage of duplicates is found within database files (69.1%) and apk files (74.8%), but this percentage is low on the other images; on another image, thumbnail files have a duplication ratio of 99.0% whereas on another image this ratio is only 0.9%.

Overall the above analysis of real-world device traces and images shows strong evidence for the potentials of deduplication on devices. They also suggest that a holistic, system-level solution is necessary to fully exploit the deduplication opportunities.

3 Design and Implementation

3.1 Design Overview

Overall the design of SmartDedup is based on the following key principles:

- I The storage on smart devices has limited bandwidth, capacity, and endurance, so deduplication should be applied as much as possible to improve its performance, utilization, and lifetime.
- II The available memory on devices is often limited, so the use of in-memory data structures should be kept as low as possible. To complement the low memory footprint, disk space should also be leveraged to keep additional data structures.
- III Many smart devices are power or energy constrained (e.g., limited battery life), and deduplication should work adaptively according to the current power or energy availability.

While following these general principles, we also cautiously design the data structures and operations used by deduplication so that its overhead is as low as possible. The rest of this section presents first an overview and then details of this design.

Deduplication can be performed at different layers (file system or block layer) of the storage stack. SmartDedup chooses the design of file system level deduplication, which allows it to exploit useful semantics and improve efficiency (e.g., avoid deduplicating unallocated blocks or processing files that have not been modified). Although hints can be passed from the file system to the block layer [16], they may not be sufficient (e.g., for providing the above semantics), and the file system's unawareness of deduplication also leads to inefficiencies. For example, the file system either cannot exploit the space saved by deduplication or has to assume a fixed deduplication ratio which does not always hold for the actual workload.

According to Design Principle I, SmartDedup considers both *in-line* and *out-of-line* deduplication to maximize the effectiveness of deduplication. In-line deduplication removes duplicate writes before they reach the disk, and can thereby quickly reduce the data and avoid the wear-out caused by duplicates. But it needs to run in the I/O path at all times; otherwise, it may miss many deduplication opportunities. Out-of-line deduplication works in the background to remove duplicates already stored on disk, and can use fingerprints stored both in memory and on disk to identify duplicates. Although out-of-line deduplication can be integrated with garbage collection to reduce wear-out [10], it is not as effective as the in-line method which removes duplicates before they reach the disk. Therefore, SmartDedup combines in-line and out-of-line deduplication to take advantage of their complementary strengths, and optimizes their uses for resource-constrained devices. In particular, these two deduplication procedures share the same fingerprint store to reduce the resource overhead (per Design Principle II); and both procedures can be dynamically enabled or disabled and dynamically change the processing rate based on a device's current power or energy status (per Design Principle III).

According to Design Principle II, to address the memory limitations of smart devices, SmartDedup adopts cohesively designed two-level in-memory and on-disk fin-

[1] Our results show that the writes to database-related files account for 21.9% of the total amount of writes, which is much lower than the 90% observed by [14]. We believe that this discrepancy is because the related work considered only writes from Facebook and Twitter, whereas we analyzed system-wide writes.

gerprint stores. The fingerprint store is the core data structure of a deduplication solution: it maintains the fingerprints of existing data so that it can determine whether new data is duplicate or not by comparing the fingerprint to existing ones. In SmartDedup, the *in-memory fingerprint store* supports fast deduplication of commonly used data with low memory cost; the *on-disk fingerprint store* keeps fingerprints that are not in memory and supports more thorough data deduplication; and fingerprints can dynamically migrate between these two stores. Together, these two fingerprint stores support the efficient operation of both in-line and out-of-line deduplication.

The rest of this section explains the various components of SmartDedup and how they function together. Our core designs, including cooperative in-line and out-of-line deduplication and tiered fingerprint stores are applicable to different types of file systems. We use our prototypes for EXT4 [18], which is the *de facto* file system on Android devices, and F2FS [15], a new flash-optimized file system increasingly used by smart devices, to explain SmartDedup.

3.2 Two-level Fingerprint Stores

In-memory Fingerprint Store. SmartDedup uses only a small amount of memory to store important fingerprints (and the corresponding PBNs) and support fast deduplication, while the other less important fingerprints are kept on disk. When the in-memory fingerprint store gets full, some of the fingerprints are demoted to the on-disk fingerprint store to make room for new ones.

To support fast fingerprint search at low memory cost, the store uses a prefix tree (implemented using a Linux radix tree [8]) as the *fingerprint index* (Figure 1). To conserve memory, different from typical indexes which provide direct locations of individual fingerprints, our index provides the locations of fingerprint groups—the fingerprints in each group shares the same prefix. For example, with an 18-bit index, all the fingerprints that share the same 18-bit prefix are grouped together. This design also facilitates the group-based fingerprint eviction and on-disk fingerprint lookup discussed later in this section. Within each group, the fingerprints are indexed by a linked list. The list is sorted by the fingerprints' remaining bits, which allows misses to be determined sooner than using an unsorted list. Moreover, the length of such a list is generally short because 1) the in-memory fingerprint store is typically small, and 2) the cryptographic hash function used for fingerprinting tends to distribute the fingerprints evenly across the different groups. Experiments from replaying our traces confirm that the average length of these lists is 9 (maximum length is 63).

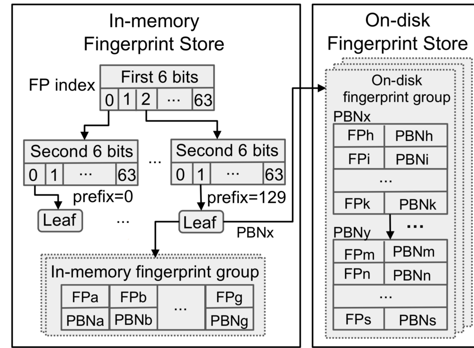


Figure 1: SmartDedup's two-level fingerprint stores.

With the above design, the space overhead of the in-memory fingerprint store is kept low. If we use 1% of the device's memory (40MB of a 4GB memory) to store MD5-based fingerprints, we can take the first 18 bits of each fingerprint as the prefix index and limit the height of the tree to three. Under this setting, the fingerprint index uses 2.03MB of memory. Considering the data structure overhead, the in-memory store can keep 1.3 million fingerprints for 5GB of unique data. For SHA1-based fingerprints, the number of fingerprints that the store can hold is 1.12 million.

The in-memory fingerprint store is used by both in-line and out-of-line deduplication as explained later. By allowing them to share this store, SmartDedup further reduces its memory usage on resource-constrained devices.

On-disk Fingerprint Store. The on-disk fingerprint store maintains the fingerprints that are evicted from memory due to the limited space of the in-memory fingerprint store. It allows SmartDedup to make full use of the existing fingerprints for providing thorough deduplication, and supports the promotion of fingerprints from disk to memory when they become important to the current workload. It is implemented as a sparse file on disk where the fingerprints are stored sequentially along with the PBNs. The size of the sparse file grows and shrinks, block by block, on demand with the number of fingerprints in the store for space efficiency. For 256GB of device storage, the total number of fingerprints that need to be stored on disk is 2^{26} in the worst case, assuming all the data is unique, which requires 2GB of disk space. In comparison, deduplicating merely 1% of 256GB of data saves 2.6GB of space and can already compensate the overhead of the on-disk store.

To enable fast search of on-disk fingerprints, SmartDedup also needs an index in memory, but it reuses the same *fingerprint index*—the prefix tree—described above for the in-memory store to reduce its memory usage (Figure 1). In fact, it adds only an address to each leaf node

of the index, which is the starting PBN of the on-disk group of fingerprints with the same prefix as the fingerprints of the in-memory fingerprint group. Each group of fingerprints is stored in an array on disk, which is sorted by its remaining fingerprint bits and stored sequentially from this PBN address in one or multiple disk blocks. In this way, the same fingerprint index is shared by both in-memory and on-disk fingerprint stores, and each leaf node can point to both an in-memory fingerprint group and an on-disk fingerprint group that share the same prefix. For a three-level tree that indexes an 18-bit prefix, the addition of the PBN in each leaf node adds at most 1MB of memory usage.

This scheme allows efficient operations on on-disk fingerprints. To search for a fingerprint on disk, SmartDedup looks for the corresponding leaf node in the fingerprint index. If the node does not exist in the prefix tree, SmartDedup knows immediately that the fingerprint does not exist on disk. If the node exists and contains a valid PBN, SmartDedup loads the whole group from that address into memory and searches for the given fingerprint in memory using binary search. The I/O overhead for accessing the on-disk groups is small because the size of each group is generally small. Assuming each group shares the 18-bit prefix, for 256GB of device storage with no duplicate data blocks, there are about 256 fingerprints per group, requiring only one to two 4KB blocks to store them. For even larger disks, we can increase the length of the prefix to bound the group size.

As discussed above, our proposed fingerprint index provides the functionality of a Bloom filter; in comparison, employing a separate Bloom filter incurs additional time and space overhead. For example, using a Bloom filter to determine whether a group of fingerprints exists or not would require 0.92MB of memory and applying five hash functions. In addition, it needs to deal with the difficulty of fingerprint deletions [7, 11].

Fingerprint Migration. Fingerprints evicted from the in-memory store are moved to the on-disk store; conversely, when an on-disk fingerprint is matched by new data, it is promoted from disk to memory. When deciding which fingerprints to evict, the in-memory fingerprint store tries to keep the fingerprints that are important to the current workload. Our evaluation results show that a simple policy such as least recently used (LRU) achieves good deduplication ratios (Section 4.2).

But the I/O overhead of fingerprint migrations is an important consideration for devices. Evicting a fingerprint from memory to disk requires two I/Os for loading the corresponding fingerprint group from disk and storing the updated group back to disk. To reduce disk I/O overhead, instead of evicting one fingerprint at a time, SmartDedup evicts a group of fingerprints at a time, so that a number of slots are freed up at once in the in-memory fingerprint store and can be used to store a number of fingerprints from the future requests. With the design of a prefix-tree-based index, the fingerprints linked to the same leaf node share the same prefix and automatically form an eviction group. Note that when migrating fingerprints from disk to memory, SmartDedup still promotes one fingerprint, instead of a whole group, at a time, since there is limited locality within each group. Moreover, with a small group size (9 on average), the deduplication ratio is also not compromised much by evicting the whole group together.

To implement a group-based eviction policy, SmartDedup keeps an LRU list for all the groups in the fingerprint index. Whenever a fingerprint is matched to a new request, its group is brought to the head of the LRU list. When eviction is needed, the entire group of fingerprints that is at the tail of the LRU list is evicted. Since both the in-memory and on-disk fingerprint stores share the same index, fingerprints that are evicted together from the in-memory fingerprint store also belong to the group that shares the same prefix in the on-disk fingerprint store, so they can be inserted into the on-disk group using a single read-merge-write operation.

3.3 Hybrid Deduplication

In-line Deduplication happens when the file system handles a write request, and it removes a duplicate write by modifying the file system's logical block to physical blocks mappings. Specifically, in our prototypes, the write paths of EXT4 and F2FS are modified in the following manner to make sure that the deduplication procedure does not violate the basic design principles of modern file systems. First, SmartDedup achieves deduplication by changing the one-to-one mappings that the file system maintains from logical blocks to physical blocks to many-to-one. Second, SmartDedup performs in-line deduplication when the file system writes back buffered data to disk; by doing so, it saves itself from processing repeated writes to buffered data, which does not hurt either performance or endurance. SmartDedup also handles direct I/Os, but the discussion here focuses on buffered I/Os since they dominate common device workloads.

In-line deduplication may not be able to find a match for a request even if there is a duplicate block on the file system, because the in-memory fingerprint store is not large enough to hold all the existing fingerprints. For such requests, in-line deduplication hands them over to out-of-line deduplication, which searches the on-disk fingerprint store in the background without slowing down the foreground application.

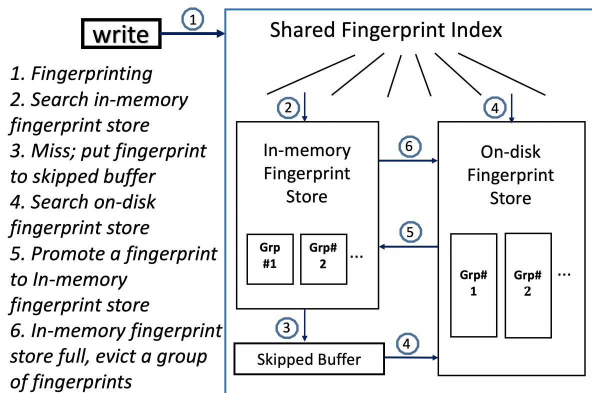


Figure 2: SmartDedup operations.

Out-of-line Deduplication works in the background on data that is not processed by in-line deduplication, which may be still in the page cache waiting to be written back or be already stored on disk. When processing a block of data, it looks for the fingerprint in both the in-memory and on-disk stores. When a match is found for a logical block, it changes the mappings between the logical block and physical block and, if needed, deallocates the redundant physical block to perform deduplication.

For efficiency, SmartDedup avoids processing data blocks that have not been modified since the last time they were processed by either in-line or out-of-line deduplication. It uses an in-memory buffer, called *skipped buffer*, which stores the list of blocks skipped by in-line deduplication, either because the latter is disabled or is enabled but cannot find fingerprint matches in the in-memory store. Each entry in this buffer stores a skipped block's information (inode number, LBN, and fingerprint (if available)) for the out-of-line deduplication to process the block quickly. It is implemented using an array-based hash table, indexed by inode and LBN. When the fingerprint of a data block is updated, the previous content of the hash table entry is replaced by the new one.

The size of the skipped buffer is kept small and is also adjustable depending on the device's current memory availability. For example, with 0.5MB of memory, the skipped buffer can store the information of 22K requests and 65K requests with and without their fingerprints, respectively. If the buffer does get full, SmartDedup converts it to store only the inode numbers of modified files so that out-of-line deduplication processes only these files. As the inode number requires only 4 bytes, the buffer rarely overflows in practice.

3.4 Putting Everything Together

Read and Write Path. When handling a write, SmartDedup fingerprints the request, searches for it in the fingerprint store(s), and deduplicates it if a match is found, as discussed above and illustrated in Figure 2. The read path in the file system is also modified to make use of deduplication to improve read performance. Because the page cache is indexed by each file's LBNs, a read that cannot find its LBN in the page cache cannot be avoided even if the requested data duplicates another logical block that is already in the cache.

To address this limitation, SmartDedup employs a *page cache index* which maps from PBNs to their corresponding pages in the page cache. For a read that cannot find a match in the page cache by its LBN, SmartDedup searches the index using its PBN, before considering it a miss as the native file system does. If a match is found, it means that the requested data already exists in the page cache, and there is no need to perform the actual read I/O. SmartDedup directly copies the data from the duplicate page in the cache. The size of this index is bounded by the size of page cache, and it can be further restricted when SmartDedup indexes only the important set of pages (e.g., the most recently used ones) in the cache.

Handling Data Updates and Deletions. A complication to the above write process is that when handling an update to or deallocation of an existing block, the fingerprint stores need to be consistently updated. SmartDedup needs to find the original fingerprint of the block so that it can update the reference count (the number of logical blocks deduplicated by this fingerprint) and delete the fingerprint if the reference count drops to zero. But SmartDedup does not have the original data's fingerprint; it has only the new request's LBN (and from there the PBN). To address this problem, SmartDedup maintains a *reverse index* (a sparse file) on disk for the fingerprint stores, which maps from a fingerprint's PBN to its corresponding leaf node in the *fingerprint index* of the fingerprint store using an array where the PBN is the index and the entries store the leaf node addresses. The leaf node represents a group of fingerprints either in memory or on disk, and SmartDedup can search this group to quickly locate the fingerprint. Compared to mapping from PBNs directly to the locations of the fingerprints, this design reduces the overhead of the reverse index because when a fingerprint migrates between the in-memory and on-disk stores, the leaf node that the fingerprint belongs to does not change and the index does not have to be updated.

Adaptive Deduplication. To further reduce the overhead of deduplication, SmartDedup can adapt its processing rate based on the current resource availability. For out-of-line deduplication, SmartDedup adapts the number of blocks that it processes per period of time (e.g., every minute). For in-line deduplication, it adapts the process-

ing rate by selectively processing n out of the N write requests that it receives—n/N defines the *selectivity*. Considering CPU and I/O load, SmartDedup automatically reduces its processing rate whenever it detects that the CPU or disk is fully utilized. Considering battery usage, SmartDedup reduces its processing rate proportionally to the remaining battery life, and completely disables deduplication when the device enters low-power mode. Similar policies for other resource constraints can also be easily specified and carried out by SmartDedup using this adaptive deduplication mechanism.

To further reduce resource usage, SmartDedup can also adapt its processing rate (by adjusting the selectivity) based on the level of data duplication observed in the workload. When the observed duplication level is low in the previous time window, SmartDedup gradually reduces its processing rate, but it quickly restores its processing rate when it detects an increasing duplication level in the current workload.

File System Consistency. Because the LBN-to-PBN mapping is already kept consistent by the native file system, the only metadata that SmartDedup needs to safekeep is the reference counts of the fingerprints—it relies on the reference counts to decide when to free a data block and when to perform copy-on-write. SmartDedup stores the reference counts persistently as part of the on-disk reverse index (together with the leaf node addresses of the corresponding fingerprints as described above).

To ensure consistency, on EXT4, SmartDedup journals the modifications to the reverse index as part of the file system journaling. The design of the reverse index helps reduce the overhead from its journaling. The entries in the index are sorted by the PBNs, so consecutive updates to the reference counts of adjacent physical blocks can be aggregated into much fewer updates to the reverse index blocks—a 4KB block stores 512 entries. Experiments using our traces confirm that the amount of additional writes to the reverse index is less than 0.5% of the total write volume. After a crash, the file system can be brought back to a consistent state by replaying the latest valid journal transaction. Similarly, on F2FS, SmartDedup ensures that modifications to the reverse index and on-disk fingerprint store are captured by the file system checkpoint so that they can always be brought to a valid state after the file system's crash recovery. The overhead of recovery is also small as it requires only updating the affected reference counts using the journal or checkpoint.

All other data structures that SmartDedup maintains can be safely discarded without affecting file system consistency. The in-memory fingerprint store will be warmed up again after the system recovers. The page cache index will be reconstructed as the page cache warms up again. The loss of the skipped buffer will make SmartDedup miss the requests that have not been processed by out-of-line deduplication. For a 0.5MB skipped buffer, at most 254MB of data will be missed (assuming that the buffer stores only the inode and LBN of each request). To reduce this impact, SmartDedup periodically checkpoints the inodes and LBNs from the skipped buffer.

	Nexus 5X	*Raspberry Pi 3*
CPU	Qualcomm Snapdragon 808	Broadcom BCM2837
RAM	2 GB	1 GB
Storage	32GB eMMC	16GB SDHC UHS-1
Operating System	Android Nougat	Raspbian Stretch Lite
Kernel Version	Linux 3.10	Linux 4.4
File System	EXT4	F2FS

Table 3: Specifications of the testing devices.

4 Evaluation

We evaluated SmartDedup based on prototypes implemented on EXT4 and F2FS. Testing devices include a Nexus 5X phone and a Raspberry Pi 3 device (Table 3). We considered the following workloads to provide a comprehensive evaluation:

- *FIO* [12]: We used FIO to create intensive I/O workloads with different access patterns and levels of duplication.
- *Trace Replay*: We replayed our collected real-world smartphone traces (Table 1), which helps us understand the performance of SmartDedup for real-world workloads of smart devices.
- *DEDISbench* [20]: We used DEDISbench to scan our collected real-world Android images and then generate workloads that reflect the data duplication characteristics (such as the distribution of reference counts) of these images.

We compared SmartDedup to two related solutions: Dmdedup [16, 23] and CAFTL [9]. Dmdedup is a block-level in-line deduplication solution that supports flexible metadata management policies. It can use a copy-on-write B-tree to store metadata and provide a consistency guarantee by flushing metadata periodically. To provide a fair comparison, we further enhanced Dmdedup by passing hints from the file system and allowing it to flush metadata only at journal commit times.

CAFTL implements both in-line and out-of-line deduplication at the flash translation layer (FTL), with several techniques designed for the resource constraints at this layer. Sampling hashing fingerprints only one data

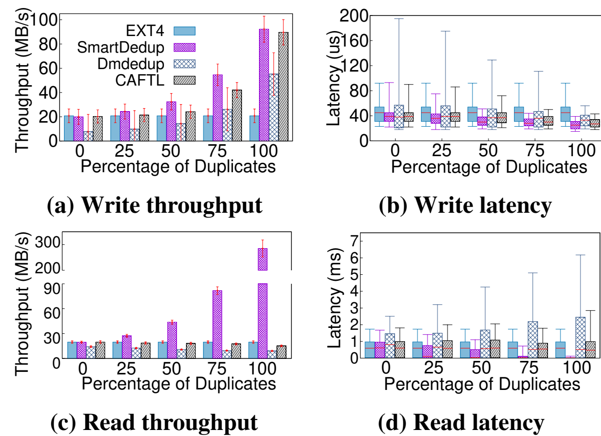
block in a request that has multiple blocks to reduce overhead. Lightweight pre-hashing applies CRC32 to pre-hash data blocks and filter out the unmatched ones to save fingerprinting overhead. Dynamic switches dynamically enable and disable deduplication based on the available cache space. For a fair comparison, we implemented sampling hashing and lightweight pre-hashing in the EXT4 writeback path, but dynamic switches are not necessary because the page cache supports rate limiting.

In all the experiments, the memory usage was capped at 3.5MB for all the evaluated solutions (unless otherwise noted). SmartDedup used 3MB for in-memory fingerprint store, which used a 14-bit prefix, and 0.5MB for the skipped buffer, which stored full information and never overflowed during the experiments. All the experiments were started with empty in-memory and on-disk fingerprint stores. Both in-line and out-of-line deduplication were used in the experiments (unless otherwise noted); adaptive deduplication was enabled and evaluated only in Section 4.4. SmartDedup uses fingerprints generated by cryptographic hash functions [19, 21] to find duplicates. The overhead for fingerprinting one 4KB block of data on Nexus 5X is about 9μs if using SHA1 and 16μs if using MD5, and on Pi is about 30μs for both. Due to limited space, we present only the MD5 results here. Each experiment was repeated at least five times.

4.1 FIO

We ran FIO with three threads, each issuing random 4KB reads or writes, using buffered I/Os (which is what real-world applications typically use). For all FIO experiments, the total read or write size was set to 2GB on Nexus and 1GB on Pi. We varied the percentage of duplicates in the workloads; at 0%, SmartDedup's in-memory fingerprint store can hold 20% and 10% of total fingerprints for the 1GB and 2GB experiments, respectively. The read experiments were performed using random reads on the data written by FIO in the write experiments (after the page cache was dropped).

Nexus 5X Results. Figure 3a and 3b show the write performance on Nexus 5X. The worst case for SmartDedup is when there is no duplicate, where SmartDedup has only 3.8% overhead in throughput and 1.1% overhead in 95th percentile latency compared to EXT4, including all the overhead from fingerprinting and operations on in-memory and on-disk fingerprint stores. In comparison, Dmdedup has a much higher overhead, 62.8% in throughput and 1.1X in 95th percentile latency, which we believe is due to 1) deduplication at the block layer adds another level of LBN-to-PBN mapping and additional overhead; 2) to guarantee consistency, the copy-on-write



(a) Write throughput (b) Write latency

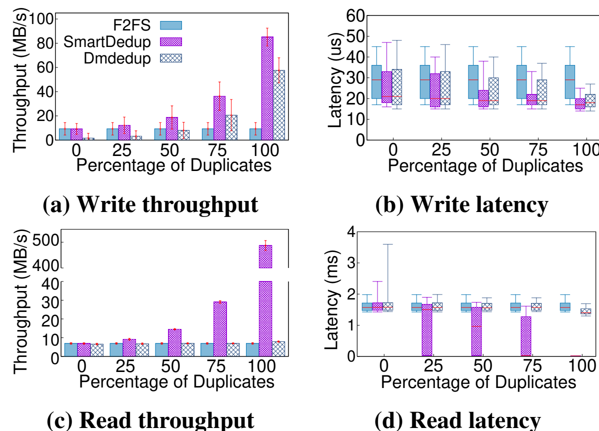
(c) Read throughput (d) Read latency

Figure 3: FIO on EXT4 using Nexus 5X. Figures 3a and 3c show the average write and read throughput, respectively, with the error bars showing the standard deviations. Figures 3b and 3d illustrate the write and read latency results, respectively, using box and whisker plots, where the whiskers indicate the 5th and 95th percentiles.

B-tree requires more metadata I/Os. For this experiment, Dmdedup introduces 75.6% more metadata writes than EXT4, whereas SmartDedup introduces only 9% more. CAFTL has less overhead than SmartDedup (1.4% less in throughput and 3.2% less in 95th percentile latency) because, with no duplicates in the workload, its pre-hashing can save substantial fingerprinting.

As the percentage of duplicates in the workload grows, the performance of SmartDedup quickly improves and exceeds EXT4. With 25% duplicates, SmartDedup already outperforms EXT4 by 16.9% in throughput and 18.5% in 95th percentile latency. Dmdedup has an overhead of 52.8% in throughput and 90.2% in 95th percentile latency. CAFTL outperforms EXT4 by only 3.1% in throughput and 6.5% in 95th percentile latency. We found out that using pre-hashing hurts deduplication performance—since requests are filtered out by pre-hashing, CAFTL does not have their fingerprints and cannot deduplicate future requests that have the same data. To verify this observation, we tried removing pre-hashing from CAFTL and the deduplication ratio indeed increases (by 14% for FIO with 25% duplicates). Without pre-hashing, CAFTL is still slower than SmartDedup (6.7% in throughput and 9.3% in 95th percentile latency), because 1) its out-of-line deduplication works only when the system is idle and cannot help much; 2) its reference-count-based eviction policy cannot exploit temporal locality in fingerprint accesses (further discussed in Section 4.2).

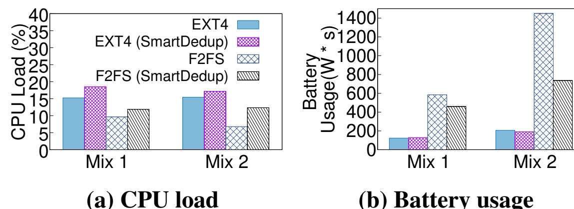
Figure 3c and 3d compare the read performance. Even in the worst case with no duplicates, the overhead of SmartDedup is small, merely 0.6% in throughput and 2.7% in median read latency, which is mainly from main-



(a) Write throughput **(b) Write latency**

(c) Read throughput **(d) Read latency**

Figure 4: FIO on F2FS using Raspberry Pi. Figure 4a and 4c represent the average write and read throughput, respectively, and the error bars represent the standard deviations. Figure 4b and 4d illustrate the write and read latency results, respectively, using box and whisker plots, where the whiskers indicate the 5th and 95th percentiles.



(a) CPU load **(b) Battery usage**

Figure 5: Resource usage of FIO.

	Write size (MB)	Duplication ratio (%)	Read/Write ratio	Source Trace #
Segment 1	17612.8	75.8	1.5	4
Segment 2	12697.6	47.9	2.2	6
Segment 3	9318.4	26.4	6.8	2
Segment 4	65.8	46.1	51.3	4
Segment 5	78.5	19.4	69.8	4

Table 4: Trace segment statistics.

taining the page cache index. As the percentage of duplicates increases, the performance of SmartDedup rises accordingly, as expected, and SmartDedup substantially outperforms EXT4, improving the throughput by up to 13.4X and reducing the 95th percentile latency by up to 93%, owing to the page cache index which allows SmartDedup to use cached data to satisfy read requests for different logical blocks but with the same content (as discussed in Section 3.4). In comparison, both Dmdedup and CAFTL are slower than EXT4. CAFTL has up to 23% and 64.1% overhead in throughput and 95th percentile latency, respectively, which we believe is due to the fragmentation induced by deduplication [13]. SmartDedup's use of page cache index helps compensate for this overhead; additional techniques [22] can also be adopted to address this problem as discussed in Section 4.5.

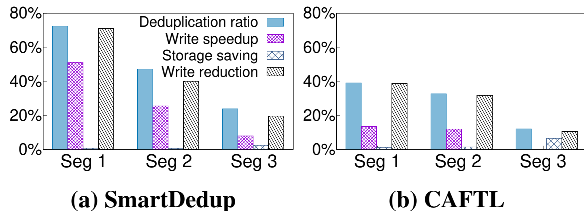
Raspberry Pi Results. Figure 4 compares the FIO performance on Raspberry Pi which has even fewer resources than Nexus 5X. The results show that SmartDedup also achieves substantial improvements compared to the native file system (F2FS) and the related solution (Dmdedup). For example, SmartDedup achieves a speedup of 31.1% and 32% in throughput for the write and read workloads with 25% duplicates, respectively compared to F2FS. Compared to Dmdedup, the throughput improvement is 2.8X and 34.5% for write and read, respectively. Although the improvement of 95th percentile latency is not as significant as in the Nexus results, SmartDedup still improves the 99.5th percentile latency substantially. Compared to F2FS, SmartDedup reduces the write tail latency by 30.4% and the read tail latency by 15.7%; and it outperforms Dmdedup by 85.4% in write and 18.7% in read.

Resource Usage. To understand the resource usage under realistic settings, we used FIO with different mixes of reads and writes to mimic the composition of real-world workloads. In our traces, the percentage of reads varies from 77.4% to 88% and the percentage of duplicates varies from 21.9% to 47.5%. For power measurement, we used the Trepn profiler [24] (for Nexus) and Watts Up Pro [5] (for Pi). Figure 5 shows the results. In Mix 1, the workload has 4GB of I/Os with 50% reads and 25% duplicates. SmartDedup's CPU overhead on EXT4 and F2FS is 3.3% and 2.2%, respectively, which are both reasonably small. For battery usage, SmartDedup has 4% overhead on EXT4 and uses 21.2% less battery on F2FS. In Mix 2, the workload consists of 6GB of I/Os with 66% reads and 25% duplicates. In this setting, SmartDedup's CPU overhead is merely 1.7% on EXT4 and 5.5% on F2FS; but it actually saves 7.7% of battery usage on EXT4 and 49.2% on F2FS because its saving on FIO runtime outweighs its overhead in power consumption. SmartDedup achieves these results while using only 0.2% (3.5MB out of 2GB) of the device's memory. Therefore, it is reasonable to believe that for typical device workloads, SmartDedup does not incur much resource overhead, and can in fact save the battery usage of the devices.

4.2 Trace Replay

The above FIO results give us insight into SmartDedup's performance and overhead under highly intensive settings. In the following two sections, we consider more realistic workloads using traces and images collected from real-world smartphones.

Replay on Real Devices. We replayed several represen-



(a) SmartDedup **(b) CAFTL**

Figure 6: Trace replay on EXT4 using Nexus 5X.

tative segments of the traces as summarized in Table 4 (Segments 1 to 3) using the real implementation of SmartDedup and CAFTL on the Nexus 5X device. These segments have different levels of duplication and all have a substantial amount of writes. Therefore, they can well represent the characteristics of the entire traces.

Figure 6 shows the results of the trace replay, including the achieved deduplication ratio and the speedup and the space and write savings compared to EXT4. The deduplication ratio reported here as well as the rest of the evaluation is computed using only the duplicates discovered by in-line deduplication. SmartDedup delivers a good speedup, up to 51.1%. It also achieves a high level of write reduction, up to 70.9%, after factoring in its own overhead in journaling and managing the on-disk data structures, e.g., 24.8 MB and 127.9 MB data were written to the reverse index and on-disk fingerprint store, respectively, during the replay of Segment 1. But there is not much space saving here, mainly because these trace segments are dominated by updates to existing data on the file system. During the replay of Segment 2, 34.2 MB and 12.5 MB disk space were used by the on-disk fingerprint store and reverse index, respectively. In comparison, CAFTL achieves less improvement (up to 13.4% and 38.7% in speedup and write reduction, respectively).

Since navigation is one of the typical applications on IoT devices [1, 2], we extracted the Google Map I/Os from our smartphone traces and replayed them (Segments 4 and 5 in Table 4) on the Raspberry Pi. As shown in Figure 7, SmartDedup also achieves good write speedup (up to 30.9%) and reduction (up to 47%) on the Pi.

Replay on Simulator. We also replayed three entire traces listed in Table 1 on a simulator of SmartDedup. The simulator implements SmartDedup's data structures and operations in user space and allows us to replay these months-long traces within a reasonable amount of time. Even though it does not model real-time performance, it allows the study of several important aspects of SmartDedup by replaying the entire traces.

First, we used the simulator to study the impact of our group-based fingerprint eviction (*G-LRU*) versus the standard, individual fingerprint eviction (*LRU*). Figure 8 shows that for different traces, *G-LRU* achieves a deduplication ratio that is at most 3% lower than *LRU*, which confirms that group-based eviction does not compromise the effectiveness of deduplication while saving substantial I/O overhead (on average 87%).

Next, we compared our recency-based fingerprint replacement, which replaces the least recently used fingerprint, to CAFTL's reference-count-based replacement, which replaces the fingerprint with the smallest reference count. As discussed in Section 4.1, pre-hashing is detrimental to deduplication ratio; here we considered the modified CAFTL that does not use pre-hashing. The results confirm the importance of exploiting temporal locality which allows SmartDedup to achieve 41.8% higher deduplication ratio than CAFTL.

We also studied the effectiveness of SmartDedup's in-memory fingerprint store design, which uses a prefix tree to index the fingerprints, by comparing its results to Dmdedup, which uses a copy-on-write B-tree as the index. We varied the amount of memory that each solution is allowed to use from 1MB to 40MB. The results show that SmartDedup's memory-conserving designs allow it to achieve higher deduplication ratios (by up to 35.7%), especially when the available memory is limited.

Finally, we evaluated the effectiveness of our two-level fingerprint store design by comparing the deduplication ratio of G-LRU with (*G-LRU*) and without (*G-LRU (in-line only)*) the on-disk fingerprint store. As expected, when the in-memory fingerprint store is small (1MB), the availability of an on-disk store and out-of-line deduplication improves the deduplication ratio from 6.7% to 12.1% (Trace 1). With a larger in-memory fingerprint store, the use of an on-disk fingerprint store still increases the deduplication ratio from 36.7% to 43.1% (Trace 2). These results prove that our designs for synergistic in-line and out-of-line deduplication with two-level fingerprint stores work well for real-world workloads, and they are particularly important for devices with limited memory capacity.

4.3 DEDISbench

In addition to using real traces, we also created additional workloads by sampling real-world smartphone images using DEDISbench [20]. We chose two of the smartphone images that we collected, with duplication ratios of 46.1% and 19.4%, and used DEDISbench to generate workloads that represent the data duplication characteristics of these images. All experiments were done on Nexus 5X using four threads and a total of 2GB of random 4KB reads or writes (SmartDedup's in-memory fingerprint store can

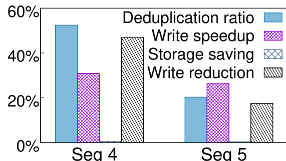


Figure 7: Trace replay on F2FS using Pi

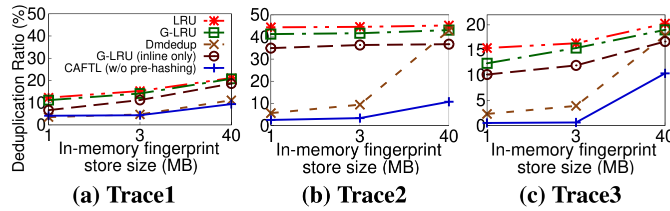


Figure 8: Deduplication ratio from different migration policies.

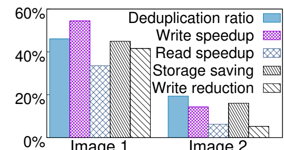
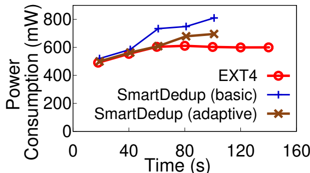


Figure 9: DEDISbench



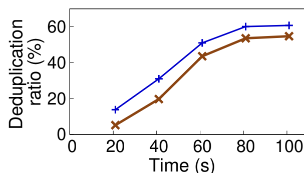
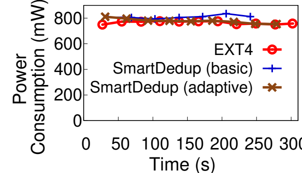


Figure 10: Adaptive deduplication based on duplication level



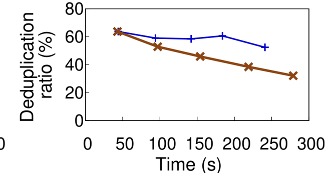


Figure 11: Adaptive deduplication based on available battery.

hold 20% of all the fingerprints during these experiments). The write experiment was done in the peak mode (the more intensive mode of DEDISbench) with the hotspot I/O distribution (which DEDISbench uses to model real-world workloads with hotspot regions in their requests). The read experiment was done by reading from what DEDISbench generated in the write experiment (after the page cache was dropped).

Figure 9 shows the deduplication ratio achieved by SmartDedup and its I/O speedups, storage savings, and write reduction compared to EXT4. The write and read speedups are both significant, up to 54.4% and 33.6%, respectively, and largely follow the deduplication ratio of the workload. The read speedup is lower than the write speedup, because not all duplicate data can be found in the page cache due to cache evictions.

The space and write savings are also substantial, up to 45.0% and 41.6%, respectively. Note that these savings are computed after SmartDedup's overhead—the space and writes used for its on-disk data structures (including the on-disk fingerprint store and the reverse index and its journal)—is factored in. These results confirm the effectiveness of our techniques (Sections 3.2 and 3.4) for reducing the I/O overhead of deduplication.

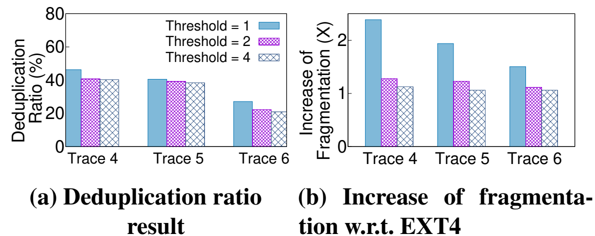
4.4 Adaptive Deduplication

Next, we evaluated the effectiveness of adaptive deduplication described in Section 3.4 by replaying trace segments on Nexus 5X. The first experiment studied the effectiveness of adapting deduplication selectivity based on the level of duplication observed in the workload. Following the general strategy described in Section 3.4, the specific algorithm used by SmartDedup is as follows. It computes the deduplication ratio of the last window—the last 150 write requests—and compares it to the average ratio from the past 30 windows. If the former is lower, it indicates that the current workload has fewer duplicates, and SmartDedup slowly reduces the percentage of requests that it fingerprints in-line in the next window (by 10% until it reaches a lower bound of 30%). But if the deduplication ratio of the last window is higher, SmartDedup quickly increases the percentage of requests that it fingerprints in-line in the next window (by 30% until it is back to 100%). How quickly SmartDedup adjusts its selectivity offers a tradeoff between performance and battery usage. We omit the sensitivity study's results due to lack of space. With the setting mentioned above, SmartDedup reduces its power consumption overhead (compared to EXT4) by up to 14% at the cost of 8% loss in deduplication ratio (as shown in Figure 10).

The second experiment evaluated adaptive deduplication based on the available battery level (Figure 11). We replayed a 12-hour long trace segment and assumed that the device's battery level was 100% (when it was fully charged) at the start of the replay and dropped to 20% (when it entered low-power mode) at the end. With adaptive deduplication, SmartDedup automatically increased the selectivity of fingerprinting as the available battery reduced. The power consumption overhead (compared to EXT4) dropped from 8%, when the battery level is 100%, to 0.3% when the battery level is 20%, at the cost of reducing the deduplication ratio from 51% to 32%.

4.5 Fragmentation Resistance

Deduplication usually brings fragmentation to disk and can hurt I/O performance. Even though flash storage is



(a) Deduplication ratio result **(b) Increase of fragmentation w.r.t. EXT4**

Figure 12: Fragmentation-resistant deduplication.

much less affected by fragmentation than HDDs, Hahn et al. [13] showed that flash devices still suffer from fragmentation due to increased I/O stack overhead. To address fragmentation, we leveraged the filtering technique from iDedup [22], which applies deduplication only to a physically contiguous sequence of writes that are duplicates. It sets a threshold on the length of a duplicate sequence, and filters out all sequences shorter than this threshold.

To evaluate the effectiveness of this filtering technique in SmartDedup, we replayed three complete traces listed in Table 1 on our simulator while varying the value of the threshold from one to four (when the threshold is one, the filtering is essentially disabled). We evaluated the impact on fragmentation, by measuring the total number of extents created by the workload. The results in Figure 12 confirm that by integrating the filtering technique, SmartDedup can reduce fragmentation without hurting the effectiveness of deduplication. For example, as the threshold increases from one to four, the deduplication ratio of SmartDedup drops by 6.0% while the increase in fragmentation (compared to native EXT4) reduces from 1.5X to 1.06X for Trace 6.

5 Related Work

There are several related deduplication solutions designed for resource-constrained systems. As mentioned in Section 4.1, CAFTL includes several techniques designed for deduplication on flash device controllers. The key differences of SmartDedup are its symbiotic use of in-line and out-of-line deduplication and the synergistic combination of in-memory and on-disk fingerprint stores for low-overhead and effective deduplication. In comparison, CAFTL relies mainly on in-line deduplication, while its out-of-line deduplication plays only a minor role and is completely separate from the former.

A recent study [17] proposed per-application, in-line deduplication for smartphones. It groups the fingerprints by applications, loads only the group for the foreground application, and swaps it out to disk when the application is switched to the background. As discussed in Section 2, per-application deduplication can miss many duplicates that exist across different applications. Moreover, migrating applications' entire fingerprint sets between memory and disk can be expensive when they become large. For example, our traces show that commonly used applications such as Gmail and Youtube have over 20MB of fingerprints and Weibo has 40MB. In comparison, SmartDedup supports system-wide deduplication with fine-grained fingerprint migration, and performs well with much lower memory usage.

Hybrid use of in-line and out-of-line deduplication has been studied in other related works. For example, DDFS [6, 25] employs both in-line and out-of-line deduplication for backup systems, and like CAFTL, they are not well integrated as in SmartDedup. DDFS caches fingerprints in memory to reduce on-disk fingerprint lookups, but unlike SmartDedup's in-memory fingerprint store, it is not designed for memory-constrained scenarios. For example, DDFS requires complex data structures to organize fingerprints that are grouped by their spatial locality. This design is important for deduplication on high-performance backup systems, but is unnecessary and costly for deduplicating the primary storage of low-end devices.

6 Conclusions and Future Work

This paper presents a deduplication solution optimized for smart devices. The novelties of this work lie in a new architectural design that synergistically integrates in-line with out-of-line deduplication and in-memory with on-disk fingerprint stores. The entire solution is cautiously designed and optimized considering the various resource constraints of smart devices. An extensive experimental evaluation based on intensive workloads and smartphone images and I/O traces confirms that SmartDedup can achieve substantial improvement in performance, endurance, and storage utilization with low memory, disk, and battery overhead. In our future work, we will further study the effectiveness of SmartDedup in other types of resource-constrained environments such as various Internet of Things and embedded storage controllers.

7 Acknowledgements

We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their thorough reviews and insightful suggestions. We also acknowledge Wenji Li and other colleagues at the ASU VISA research lab for their help in collecting the traces. This research is sponsored by the National Science Foundation CAREER award CNS-1619653 and awards CNS-1562837, CNS-1629888, IIS-1633381, and CMMI-1610282.

References

[1] Android auto. https://www.android.com/auto/.

[2] Apple car play. https://www.apple.com/ios/carplay/.

[3] Apple file system (APFS). https://developer.apple.com/wwdc/.

[4] VISA lab traces. http://visa.lab.asu.edu/traces.

[5] Watts Up Pro power meter. https://www.vernier.com/products/sensors/wu-pro/.

[6] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer*, 50(7):64–72, 2017.

[7] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: From I/O ports to process management*. O'Reilly Media, Inc., 2005.

[9] 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 *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[10] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 29–44, Santa Clara, CA, 2017.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[12] FIO — Flexible I/O tester synthetic benchmark. http://git.kernel.dk/?p=fio.git.

[13] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 759–771, Santa Clara, CA, 2017.

[14] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 309–320, 2013.

[15] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[16] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 315–322, 2016.

[17] Bo Mao, Suzhen Wu, Hong Jiang, Xiao Chen, and Weijian Yang. Content-aware trace collection and I/O deduplication for smartphones. In *Proceedings of 33rd International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[18] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new EXT4 filesystem: Current status and future plans. In *Proceedings of the Linux symposium*, pages 21–33, 2007.

[19] FIPS PUB NIST. 180-1: Secure hash standard, 1995.

[20] Joao Paulo, Pedro Reis, Jose Pereira, and Antonio Sousa. DEDISbench: A benchmark for deduplicated storage systems. In *Proceedings of Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 584–601. Springer, 2012.

[21] Ronald Rivest. The MD5 message-digest algorithm. RFC 1321, Internet Request For Comments, 1992.

[22] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2012.

[23] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of Linux Symposium*, 2014.

[24] Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler/.

[25] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2008.

Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!

Abhinav Duggal Fani Jenkins Philip Shilane Ramprasad Chinthekindi Ritesh Shah
Mahesh Kamat
Dell EMC

Abstract

Data Domain has added a cloud tier capability to its on-premises storage appliance, allowing clients to achieve the cost benefits of deduplication in the cloud. While there were many architectural changes necessary to support a cloud tier in a mature storage product, in this paper, we focus on innovations needed to support key functionality for customers. Consider typical customer interactions: First, a customer determines which files to migrate to the cloud by estimating how much space will be freed on the on-premises Data Domain appliance. Second, a customer transfers selected files to the cloud and later restores files back. Finally, a customer deletes a file in the cloud when its retention period has expired. Each of these operations requires significant architectural changes and new algorithms to address both the impact of deduplicated storage and the latency and expense of cloud object storage. We also present analysis from deployed cloud tier systems. As an example, some customers have moved more than 20PB of logical data to the cloud tier and achieved a total compression factor (deduplication * local compression) of 40× or more, resulting in millions of dollars of cost savings.

1 Introduction

Today many customers want options to migrate portions of their data to cloud storage. Object storage in public and private clouds provides cost-effective, on-demand, always available storage. Data protection is a key requirement, and Data Domain [32] has traditionally served as an on-premises data protection product, holding backups of customers' primary data. Data Domain added a deduplicated *cloud tier* to its data protection appliances. Our deduplication system consists of an *active tier* where customers backup their primary data (typically retained for 30-90 days) and a cloud tier where selected backups are transitioned to cloud storage and retained long term (1-7 years). Our recent cloud tier product is currently being used by hundreds of customers.

Adding a cloud tier to a mature storage appliance involved numerous architectural changes to support local and remote storage tiers. We present some of the most novel improvements necessary to support basic capabilities required by customers. First a customer wishes to free up space on their active tier by migrating files to the cloud and wishes to determine how much space will be saved (Section 4). While this is straightforward to calculate in traditional storage systems, deduplication complicates the process because the files may have content that overlaps with other files that will remain on the active tier. We present a new algorithm to estimate the amount of space unique to a set of files. This algorithm builds upon a previous technique using perfect hashes and sequential storage scans [7] for memory and I/O efficiency.

Once a customer selects files for migration to the cloud tier, we wish to transfer the unique content to the cloud tier to preserve the benefits of deduplication during transfer. When a large quantity of data is being initially transferred, we developed a bulk seeding algorithm that also uses perfect hashes to select the set of chunks to transfer (Section 5). For ongoing transfers, we developed a file migration algorithm that leverages metadata that is stored locally to accelerate the deduplication process and avoids the latency of accesses to cloud object storage (Section 6.1). We then describe how files can be efficiently restored to the active tier.

Finally, as a customer deletes files from a cloud tier, unreferenced chunks must be removed to free space and reduce storage costs (Section 7.2). While garbage collection for the active tier has been described [11], we updated the cloud tier version to handle the latency and financial cost of reading data from the cloud back to the on-premises appliance.

From experience with a deployed cloud tier, we have learned lessons about sizing objects stored in private and public object storage systems and trade-offs of performance and cost. After analyzing deployed systems, we found that customers achieve a range of deduplication ratios. Our customers achieved an active tier deduplication ranging between 1× and 848× and cloud tier deduplication ranging between 1× and 66×. The space savings result in cost savings and one customer saved as much as $10 million (Section 8).

Our largest system has a single 1PB active tier and two 1PB cloud units within a single cloud tier. This is the physical capacity before the benefits of deduplication and compression. A cloud unit is a single deduplication domain. Each cloud unit has its own metadata, data and fingerprint

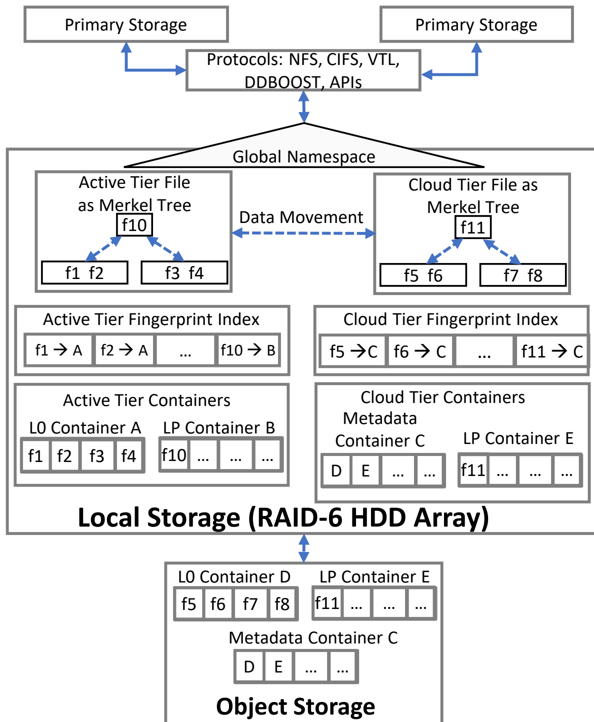


Figure 1: Data Domain with active and cloud tiers

index. For simplicity, we will use *active tier* and *cloud tier* as the terminology when referring to the active and cloud components of our backup appliance. Our cloud tier is designed to work with object storage that is either on-premises or in a public cloud, and we use the terms *object storage* and *cloud storage* interchangeably.

The vast majority of the technology described in this paper is publicly available and used by our customers. The exception is an experimental improvement to our garbage collection algorithm described in Section 7.2 based on microservices for public cloud providers. We believe the following contributions, except the performance results, are applicable to other deduplicated storage systems adding a cloud tier.

1. An architecture for a deduplicated cloud storage tier
2. A space estimation algorithm for files within deduplicated storage
3. Algorithms to seed a cloud tier or perform ongoing file migration
4. A garbage collection algorithm designed for cloud storage properties
5. An evaluation of how customers are using the cloud tier
6. Performance results on internal systems

2 Architecture

We updated our deduplicated storage architecture to support both a local, active tier and a remote, cloud tier. We first review the active tier architecture and then describe changes for a cloud tier as shown in Figure 1.

2.1 Active Tier Architecture

A file in both, active and cloud tiers, is represented by a Merkle tree with user data as variable sized chunks at the bottom level of the tree, referred to as *L0 chunks*. The SHA1 fingerprints of those chunks are grouped together at the next higher level of the tree to form chunks, referred to as L1 chunks. SHA1 fingerprints of L1 chunks are grouped together as L2 chunks, and this continues up to L6 which represents the entire file. The top chunk of the tree is always an L6 chunk, even though it may refer to chunks in a lower numbered level. We refer to chunks above L0 as LP chunks. The L6 chunk of every file is stored in a namespace which is represented as a B+ Tree [11].

Deduplication happens when different files refer to the same L0 and LP chunks. As an example, if two files are exactly the same, they would have the same L6 fingerprint. But if two files only partially overlap in content, then some branches of the tree will be identical (LP and L0 chunks), while other branches will have different fingerprints. Multiple L0 chunks are compressed into 64K-128K sized compression regions, while LP chunks are not compressed as SHA1 fingerprints are quite random and do not compress. If encryption is enabled, the compression regions are also encrypted before they are written to storage containers.

A Data Domain appliance tends to be utilized by a single customer, who typically selects a single encryption key for all of the data. If multiple keys are selected, customers accept a potential loss in cross-dataset deduplication. We have not found customer demand for convergent encryption or stronger encryption requirements for cloud storage than on-premises storage.

All chunks (L0s and LPs) are written into storage containers. On the active tier, containers are 4.5MB in size, while the container size on the cloud tier varies with properties of each cloud storage system (Section 2.3). We segregate the L0 and LP chunks into separate containers, which we refer to as *L0-Containers* and *LP-Containers*, respectively. Creating separate LP-Containers supports various operations like garbage collection that need to process LP chunks. Segregating L0 chunks from LP chunks also ensures that the locality of L0 chunks is preserved which results in better read performance. Both types of containers consist of a data section with the chunks and a metadata section with fingerprints of the chunks. During deduplication, container metadata sections are loaded into a memory cache. This loads 1,000 or more fingerprints into memory, helps to accelerate deduplication, and reduces fingerprint index accesses from disk. During reads, container metadata sections are loaded into the same memory cache and this avoids having to read all subsequent fingerprints in the container from disk [32].

2.2 Cloud Tier Architecture

With the introduction of cloud tier, there is a single namespace, referred to as global namespace. The global namespace spans both, active tier and cloud tier files. The Merkle trees of files in the cloud tier are stored on the local storage of the Data Domain system to facilitate high-speed access of those files. The global namespace which contains the L6 chunks of every file is periodically written to the cloud tier. In this design, files exist in one tier or the other based on a customer's policy. In some cases, customers use the cloud tier as extra capacity, while other customers use it for long term archival of selected data.

For the cloud tier, we introduced a third type of container, which we refer to as *Metadata-Container*. Metadata-Containers store the metadata sections from multiple L0 and LP-Containers. Since the metadata sections of containers are read during deduplication and garbage collection, and require quick access, Metadata-Containers are stored on the local storage as well as in cloud storage.

The SHA1 fingerprints of chunks are stored in an on-disk fingerprint index which consists of a mapping of fingerprint to container number. To avoid these writes and reads to the cloud, the cloud tier fingerprint index is stored on the local storage of the Data Domain system. In the active tier, the fingerprint index contains fingerprint to L0 or LP-Container number mappings, while the cloud tier index contains fingerprint to Metadata-Container number mappings. This is an optimization to load fingerprints from a local Metadata-Container instead of a remote L0 or LP-Container.

Figure 1 shows an active tier file that contains L0 chunks with fingerprints f1, f2, f3, and f4. The L0 chunk fingerprints are stored in an LP chunk with fingerprint f10. The L0 chunks and their fingerprints are stored in an L0-Container A, while the LP chunks and their fingerprints are stored in an LP-Container B. The active tier fingerprint index contains mappings for all the L0 and LP chunks' fingerprints. Figure 1 also shows a cloud tier file. The global namespace contains the location of this file. The file contains L0 chunks with fingerprints f5, f6, f7, and f8. The L0 chunk fingerprints are stored in an LP chunk with fingerprint f11. The L0 chunks and their fingerprints are stored in an L0-Container D, while the LP chunks and their fingerprints are stored in an LP-Container E. L0-Container D is written to object storage in the cloud, while LP-Container E is written to both, object storage in the cloud and local storage. The cloud tier fingerprint index contains mappings for all the L0 and LP chunks' fingerprints, and is stored on local storage.

As explained above, we store critical cloud tier metadata on the local storage of the Data Domain system to improve performance and reduce cost. The majority of this metadata, including the global namespace, is also mirrored to the cloud in order to provide a disaster recovery (DR) functionality. Disaster recovery is needed if a Data Domain system with a cloud tier was lost in a disaster. Such a disaster can result in the loss of active tier and the local cloud tier storage, where cloud tier metadata resides. Disaster recovery is the main reason why the active tier and cloud tier have different deduplication domains. If an active tier is lost, the backup copies migrated to object storage can be recovered.

The DR procedure to recover the cloud tier includes procuring a replacement Data Domain system and initiating an automated recovery process, which involves: creating a new cloud unit, copying the metadata (LP and Metadata-Containers) to the local storage for the cloud tier, rebuilding the cloud tier fingerprint index, and recovering the global namespace. Note that we only recover the metadata from the cloud storage. The data continues to reside in cloud storage, and it is not copied to the Data Domain system. After the DR procedure is completed, the cloud tier is accessible from the new Data Domain system.

To summarize our new architecture, we support an active tier and a cloud tier. Both tiers offer the benefits of deduplication, however each tier is a separate deduplication domain. Chunks are not shared between the active tier and the cloud tier. Note that if the underlying object storage provides deduplication, it would be ineffective since our deduplication algorithm removes the majority of duplicates at 8KB granularity. A global namespace maintains the location of both, active tier and cloud tier files. Each tier has its own metadata, data, and fingerprint index. To eliminate the costly reads and writes from/to the cloud, we copy key cloud tier data structures on the local storage of the Data Domain system. The data structures stored locally are the Metadata-Containers, the LP-Containers, the Merkle trees, and the cloud tier fingerprint index. The Metadata-Containers and LP-Containers are also mirrored to the cloud to facilitate disaster recovery.

2.3 Object Sizes for Cloud Tier

The cloud tier architecture we have chosen allows us to select an optimum object size because we write L0-Containers as individual objects and we have the ability to control the container size, up to 4.5MB. We are not able to write containers larger than 4.5MB without significant changes to our implementation. In the case of cloud tier, we use the terms objects and containers interchangably.

We started with 64KB objects, but evolved to larger sizes in the range of 1-4MB for several reasons. Larger objects result in less metadata overhead at the cloud storage provider because they store per-object metadata. Larger objects also decrease transaction costs as cloud storage providers charge per-object transaction costs. We have also discovered that larger objects perform better. Using an internal tool, we experimented with object sizes ranging between 64K and 4MB. On private cloud storage such as ECS [8], we saw a 6x improvement with 4MB objects compared to 64KB objects. On public cloud storage such as Amazon S3 [3], we saw a 2x

improvement with 4MB objects compared to 64KB objects. Even though the performance does not improve much after 1MB, a higher object size is better as it reduces the cost of transfer. Ultimately, writing larger-sized objects is a better choice for our cloud tier solution, though the exact size we choose varies with properties of each cloud storage system. In some cases certain object sizes align better with the provider's block size. For different providers, we choose different object sizes ranging from 1MB to 4MB.

3 Background on Perfect Hashing and Physical Scanning for the Cloud Tier

In addition to the architecture changes described in Section 2.2, we introduced several new algorithms specific to cloud tier. Since these algorithms utilize the perfect hashing technique from Botelho et al. [7] and the physical scan technique of Douglis et al. [11] as building blocks, we briefly review those works. Perfect hashing and physical scanning provide the basis for building the following cloud tier algorithms: space estimation (Section 4), seeding (Section 5), and cloud tier garbage collection (Section 7.2). Adapting perfect hashing and physical scanning to these cloud tier algorithms was mostly an engineering effort, and the novelty is specific to solving the challenges of the cloud tier algorithms and not necessarily the underlying techniques used.

3.1 Perfect Hashing

For algorithms described below, we need to perform a membership query for our fingerprints. Perfect hashing is a technique that helps us to perform this membership query by representing a fixed key set. We use a *perfect hash* vector which consists of a perfect hash function and a bit vector [7, 11]. A perfect hash function is a collision-free mapping which maps a key to a unique position in a bit vector. To generate the hash functions, a static set of fingerprints under consideration is used to generate a 1:1 mapping of fingerprint to a unique position in the bit vector. The function building process involves hashing the fingerprints into buckets where multiple fingerprints map to one bucket. Then for the fingerprint set in each bucket, we build a perfect hash function. By dividing the fingerprints into these buckets, we can build functions for each bucket in parallel. Once we obtain the function for each bucket, we store the function in a compact way [5, 6, 9]. Without the compactness of the perfect hashing representation, we would not have sufficient memory to reference all fingerprints in the system.

Building the perfect hash functions is quite efficient. For example, on a full 1PB system, we can build the perfect hash functions for 256 billion fingerprints in less than 3 hours with about 2.8 bits needed per fingerprint. The bit value in the perfect hash vector is used to record membership, such as chunk liveness for garbage collection.

3.2 Physical Scanning

As files are deduplicated, new LP chunks are written which refer to lower level LP and L0 chunks. Say two L2's written by two different files refer to the same L1, then these L2s most likely will get written to different containers. Hence these LP chunks get fragmented over time. For algorithms described in the sections below (e.g. garbage collection), we need to walk the LP chunks of all or most of the files in the system.

One way to walk the Merkle trees of LP chunks is to do it in a depth first manner. For every file, walk from L6 chunk, to L5, L4, L3, L2 and down to L1 chunks and get the L0 refererences. There are two problems with this traversal. First is that if two files point to the same LP chunk, then by doing file by file walk in depth-first manner, we will enumerate the same LP twice. The second problem is that since these LP chunks can be in different containers, loading these LP chunks will result in doing a random lookup to first get the location of LP chunk from the fingerprint index and second to read the LP chunk. Over time as deduplication increases, the same LP and L0 chunks get referenced multiple times and the LP fragmentation worsens.

Hence, instead of doing an expensive depth-first traversal, enumeration is done in a breadth first manner. By keeping track of LP chunks we have already enumerated (using a perfect hash vector), we avoid enumerating the same chunk twice. To reduce the random lookups, we first segregated LP and L0 chunks in different containers. By doing this, we converted random lookup for every LP to a random lookup for a group of LPs present in same compression region of the container. Here is a summary of the steps we follow to perform a physical scan of the file system:

Analysis Step: We walk the on-disk fingerprint index to create three perfect hash vectors as described in Section 3.1. Two of the perfect hash vectors are called the *walk vector* and *read vector*, respectively, and are used to assist in the breadth-first walk of the Merkle trees of all the files. These two vectors are only built for LP chunks. The third vector is called the *fingerprint vector*, and is used to record the liveness of a fingerprint. The fingerprint vector has bits for LP and L0 chunks. 97% of chunks are L0s, so the fingerprint vector is the largest.

Enumeration Step: We perform a number of sequential scans of containers to find chunks at specific levels in the Merkle trees. We first walk the namespace and mark all L6 fingerprints in the walk and fingerprint vectors. The top chunk is always a L6 chunk, which may refer to any lower-numbered chunk. Our system has an in-memory structure that records which LP types (L6, L5, ... L0) exist in each container, so we can specifically scan containers with L6 chunks. Figure 2 shows the next two steps of the enumeration process along with how the perfect hash vectors are used. Blue indicates the state of perfect hash vectors in the

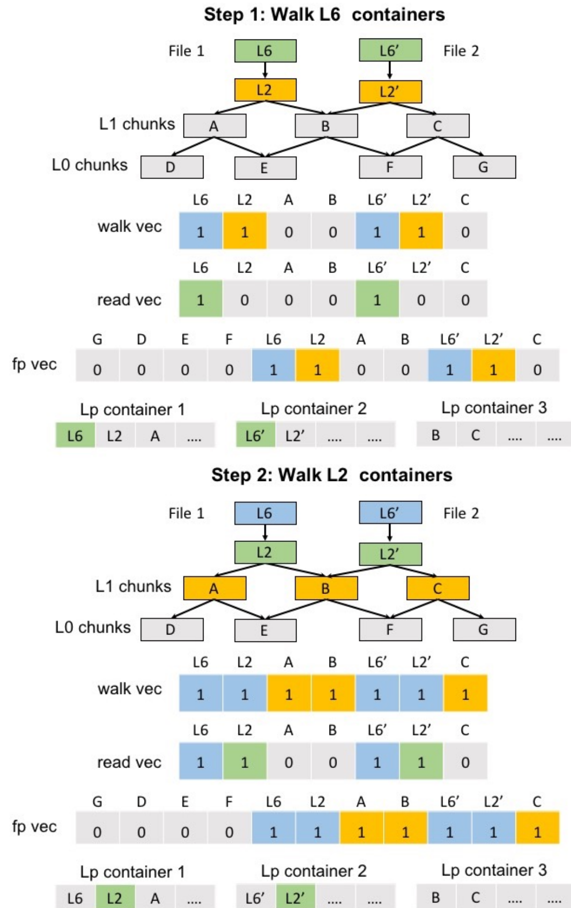


Figure 2: Physical Enumeration process

previous step, green indicates that we are reading containers and setting the read vector and yellow indicates that we are setting the walk and fingerprint vectors and gray indicates that we are yet to process those fingerprints. In step 1, we walk the container set reading L6 chunks and for any L6 fingerprints marked in the walk vector, we mark that L6 in the read vector (green). In this example, the L6 chunks reference L2 fingerprints, so we mark the L2 fingerprints in the walk (yellow) and fingerprint vectors (yellow). In step 2, we walk the container set again reading L2 chunks and for any L2 fingeprints which are marked in walk vector, we mark those L2 fingerprints in read vector (green). We read the L1 fingerprints from the marked L2 chunks and mark those L1 fingerprints in the walk (yellow) and fingerprint (yellow) vectors. For deeper trees, we repeat the steps of reading a level, marking fingerprints in a walk vector and fingerprint vector, and then reading the lower level and marking fingerprints in the walk vector. Finally, in the L1 container set walk, the L1 chunks have a list of L0 fingerprints, which we mark in the fingerprint vector. Parallel I/O is leveraged to read containers from the RAID array, and multiple threads are used for marking the chunks in the walk, read, and fingerprint vectors.

4 Estimate Freeable Space

Deduplication creates a new challenge for customers interested in reducing their active tier footprint. In traditional storage, transferring 10GB of files would reduce the active tier by 10GB. With deduplicated storage, less space may be freed because of content overlap with files that remain on the active tier. So, a customer may end up paying for on-premises capacity as well as object storage capacity despite their intention.

Such customers need a way to evaluate how much space would be freed on the active tier by moving files to the cloud tier. Variants of this problem have been considered for directing content to storage nodes [10, 12, 20] to maximize deduplication and to evaluate the unique space referenced by volumes of block storage [14]. Our system does not maintain reference counts due to the difficulty of maintaining accurate counts under complex error cases, so we implemented an algorithm to calculate the space that would be saved on the active tier if selected files were migrated to the cloud.

Using the perfect hashing (Section 3.1) and physical scanning (Section 3.2) techniques, we walk the files selected for migration in a breadth-first manner through the Merkle tree and mark the chunks in an in-memory perfect hash vector. Then, we walk all the remaining files (those not selected for migration) and repeat the breadth-first traversal again. In the second traversal, we unmark the chunks referenced by these remaining files. After this traversal is complete, the chunks which are still marked in the perfect hash vector are the chunks which are uniquely referenced by files selected for migration to the cloud tier. We perform a walk of the containers and sum up the chunk sizes for any chunks marked in the perfect hash vector.

This gives us an exact count of the bytes that would be freed. It becomes an estimate when new files are written to the active tier after constructing the perfect hash vector and such files deduplicate with selected chunks. In such cases, our algorithm overestimates the number of bytes that would be freed due to migration.

Since space estimation uses perfect hashing, the estimation cannot be done until the perfect hash functions are generated. As discussed in Section 3.1, this process takes nearly 3 hours for 1 PB of physical data. Also, this process gives only a point in time space estimate. As new data gets written to active tier, it can deduplicate against the chunks that will be moved to the cloud storage. As a result the point in time estimate becomes stale. Our customers can run this tool periodically to get the updated estimate. In practice, this has not been an issue.

5 Seeding

When customers started using cloud tier, they faced new challenges with data migration to the cloud. Some customers

had hundreds of TBs of data on the active tier that they wanted to migrate to the cloud tier. Being a deduplication system, migration of large amount of data to the cloud suffered from the same challenges as described in Section 3.2. Hence, we implemented a seeding algorithm based on the perfect hashing (Section 3.1) and physical scanning (Section 3.2) techniques.

We first build an in-memory perfect hash vector for fingerprints in the active tier fingerprint index. Then, for all the files that need to be transferred to the cloud tier, we traverse the Merkle trees in the same breadth first manner and mark the corresponding chunks live in the perfect hash vector. We then walk the containers on the active tier, pick the chunks which are marked live in the perfect hash vector, pack them in destination containers, and write them to cloud tier. This process generates both data containers (L0) and containers for metadata (LP and Metadata-Containers). Once all these containers are written, we update the namespace to indicate that all these files are now located in the cloud tier.

Seeding needs to be resumable because the memory requirements for seeding and GC are high and memory is shared between these processes. Cloud GC and seeding share memory, but until seeding is complete, there is no need to run cloud GC since the cloud tier is nearly empty. Active GC and seeding may need to run at the same time, especially since seeding may take weeks to transfer data depending on WAN speed. To make it resumable, after we mark the chunks that need to be transferred in the perfect hash vector, we persist the perfect hash vector (function and bit vector) to disk and then start active GC. Once active GC finishes, we load the perfect hash vector and resume writing to the cloud.

Seeding guarantees that all the L0 and LP chunks are transferred to the cloud tier before the file's location is changed in the namespace. It uses the perfect hash vector to guarantee that property. As we transfer chunks to the cloud tier, we change the membership bit in the perfect hash vector from 1 to 0. At the end of seeding, all the bits in the perfect hash vector should be 0 to guarantee that all necessary chunks are transferred.

There is an important caveat with seeding. The chunks written to the cloud tier do not pass through the deduplication process. So, seeding is only used to transfer large amounts of data for the first time when a customer buys a cloud tier license. We cannot use the typical fingerprint caching and prefetching approach for deduplication because the seeding process works in physical disk order instead of the logical order of chunks in a file, so caching is ineffective. Another approach is to consider generating a perfect hash vector for data already stored in the cloud tier, but this would double the memory requirements since we would need to generate perfect hash vectors for both the active and cloud tiers. We already use a large perfect hash vector and cloud tier had to be supported on existing active tier systems in the field, so adding more memory for this is not a practical solution.

6 File Migration and Restore

6.1 File Migration

Unlike seeding which is a one time process to transfer a large amount of data from the active tier to a nearly empty cloud tier, file migration is designed to transfer a few files incrementally. File migration reduces the amount of data transferred to the cloud tier by performing a deduplication process relative to chunks already present in the cloud tier.

File migration starts in the active tier by traversing the Merkle trees for selected files in a depth-first search manner. This traversal is performed in parallel for a number of selected files. The traversal ends when we reach the desired L1 chunks, which contain L0 fingerprints. The fingerprints are checked against the cloud tier using the same container metadata prefetching technique used during deduplication to see if identical fingerprints are already present in the cloud tier. If the fingerprints are not present in the cloud tier, then the corresponding chunks are read from the active tier and packed to form new L0-Containers. This process generates both data containers (L0) and containers for metadata (LP and Metadata-Containers). After Metadata-Containers are written, all contained fingerprints are added to the cloud tier fingerprint index. Finally, the namespace is updated to indicate that the file is located in the cloud tier.

To compare seeding and file migration, seeding is designed for bulk transfers such as when first moving files from a full active tier to nearly empty cloud tier. Seeding has the overhead of generating the perfect hash function, which is nearly 3 hours per 1PB of physical capacity. This is acceptable relative to the many days or weeks possibly required to transfer a large dataset to the cloud. Seeding has the advantage of physical enumeration, which uses sequential I/O instead of random I/O. On the other hand, when transferring a small number of files, it is more efficient to perform the random I/O to find the needed chunks for the files than generate the perfect hash function. Newer backup files on the active tier also tend to have better physical locality as our deduplication engine explicitly writes duplicates to keep locality high [2, 25]. Seeding and file migration are experimentally compared in Section 8.

6.2 File Restore

Restoring a file back from the cloud tier involves the reverse process of reading the Merkle trees to the L1 chunks in a depth-first manner. The L1 chunks are read from local storage since the LP-Containers are stored locally. The L0 fingerprints are checked against the active tier fingerprint index, and if present, do not need to be read from cloud tier. For new L0 fingerprints (not present in the active tier), we get the Metadata-Container number from the cloud tier fingerprint index, read the Metadata-Container to get the L0-

Container number, read the L0-Container from object storage, and write the relevant L0 chunks to L0-Containers in the active tier. As L0-Containers are being written to active tier, new Merkle trees for the active tier are formed in a bottom-up manner. This process generates new LP chunks which are written to LP-Containers in the active tier, and the namespace is updated.

To summarize, while performing file migration to the cloud tier, we deduplicate against chunks in the cloud tier, and while restoring files from the cloud tier, we deduplicate against chunks in the active tier.

7 Garbage Collection (GC)

When customers expire backups, some chunks become unreferenced. The Data Domain filesystem is log-structured and garbage collection is responsible for cleaning the unreferenced data asynchronously. Our garbage collection is mark and sweep based and described in our previous work [11]. To briefly summarize the process, we use the physical scanning technique described in Section 3.2 to mark chunks live (referenced from live files) in the perfect hash vector. Then, we perform the sweep operation. Active tier GC and Cloud tier GC differ in the sweep process, and we describe the differences here. Most of our customers run GC on the active tier weekly according to a default schedule, whereas cloud tier garbage collection varies from once every two weeks to once every month to never.

7.1 Active Tier Garbage Collection

After marking chunks live in the perfect hash vector, the sweep process walks the container set to copy live chunks from old containers into newer containers while deleting the old containers. This sweep process is also called copy forward. This process focuses on a subset of the container set which will give us the maximum space back.

During this process we first read the source containers from disk. Then, we check the fingerprints from the metadata sections of these containers against the perfect hash vector to determine which L0 chunks are live. Finally, we decrypt (if encrypted) and uncompress the compression regions inside the source containers, encrypt and compress the live chunks into new compression regions, and pack them into destination containers which are written to disk. The copy forward process for LP Containers does not perform decompression/-compression since LP chunks are not compressed.

7.2 Cloud Tier Garbage Collection

As data is written to the cloud tier, space usage and costs grow. Similar to the active tier, when customers expire files in the cloud, GC needs to clean unreferenced chunks on the cloud tier. The challenge for cloud GC is that L0 containers are not local and reading them from the cloud is expensive. It is also slow in terms of performance as we have to read the container objects over WAN.

From our experience with active tier, we know that a single cycle of GC copies forward 15% of the containers on average, where each container has an average of 50% live data. Hence, for 1PB of physical capacity in a cloud tier, we need to read 150TB of partially-live containers and write 75TB of newly-formed containers, increasing data transfer costs and transactional costs. As an example, based on AWS pricing, we calculated the cost of copy forward for 150TB of data to be nearly $14,000 per GC cycle. The major contributor to this cost is egress data transfer cost, so we needed a way to do garbage collection without reading the L0-Containers from object storage and writing new L0-Containers to object storage. To address this, we defined an API to perform copy forward inside the cloud provider's infrastructure. ECS and VirtuStream cloud providers have implemented this API. The API takes a list of source container objects, a list of offset ranges inside of those container objects, and a destination container object. The offset ranges correspond to live compression regions within the source container objects. Upon receiving the API, the cloud provider copies the live offset ranges from the source container object to a new, destination container object.

In order to perform the chunk level copy forward done by active tier GC, the compression regions need to be decrypted and uncompressed, and the live chunks need to be copied forward to form new compression regions inside new containers. Doing all of this in the cloud provider's infrastructure poses a challenge as we need to expose our container format, compression libraries, and encryption keys to the cloud provider. To address this, the new API we have implemented does not decompress/compress or decrypt/encrypt compression regions. It performs copy forward at the compression region level. We only delete a compression region in a cloud container when it is completely unreferenced. This approach is different from active tier GC where we delete individual chunks. The advantage of treating an entire compression region as live or dead is that the service running inside the cloud does not need to understand the container format or to compress/uncompress or decrypt/encrypt the compression regions. The service just reads offset ranges of where the live compression regions reside in existing container objects and writes a new destination container object. The disadvantage is that we won't delete a compression region until all the chunks inside the compression region are unreferenced. This can reduce our cleaning efficiency. In Section 8.1.2, we present field analysis of how live and dead data is distributed in compression regions and how much cleaning efficiency is lost due to compression region based cleaning instead of chunk based cleaning.

The cloud GC copy forward process works as follows. Once the live chunks have been identified in the per-

fect hash vector using the physical scanning technique described in Section 3.2, cloud tier GC performs a copy forward process of the Metadata-Containers. This process copies the live chunks' metadata from an existing Metadata-Container to a new Metadata-Container, and deletes the old Metadata-Container. The copy forward process of Metadata-Containers occurs on the local storage as Metadata-Containers are stored locally. As part of this process, we create a recipe containing the old L0-Container, the offset ranges of the live compression regions in that L0-Container, and a destination L0-Container which is being generated. Next, we send this recipe to the cloud provider in the form of the new API, and the cloud provider performs the copy forward of compression regions within their infrastructure. We then delete the old container objects.

For public cloud storage like AWS, Azure, and Google Cloud, such an API does not yet exist, so we have created an experimental version (not yet available to our customers) using a microservice that can be deployed and run inside the public cloud provider environment. Our plan is to work with cloud providers to either use a custom API for copy forward or our microservice. The results from the API and microservices based approaches are presented in Section 8.

8 Evaluation

We divide our evaluation into results from deployed systems and results from internal experiments.

8.1 Deployed Systems Evaluation

In this section, we show how our customers are using cloud tier in terms of data written, deduplication, and data deleted. We do not present results for space estimation and seeding in this section as these were recently released and the sample set is statistically small. Those techniques are evaluated on internal systems. We considered results from hundreds of deployed systems, filtered out systems with less than 1TB (post dedup/compression) of data in the cloud tier and inconsistent reports, and randomly selected 200 systems for analysis. Across systems, the age of the cloud tier varied from a few months to over two years.

Figure 3 shows monthly cumulative bytes (before deduplication and compression) sent from active tier to cloud tier for our 200 systems in the last three months. On one end, in 35% of the cases, less than 60TB (before dedup/compression) per month has been moved to cloud tier. On the other end, in 15% of the cases, over 660TB per month was sent to the cloud tier.

Figure 4 shows a scatter plot comparing the amount of data (before deduplication and compression) the deployed systems have moved to cloud tier and the total compression they have achieved. Total compression ratio is defined as deduplication ratio * local compression ratio. The general trend is that as more data is moved to the cloud tier, the more total compression is achieved because the new data deduplicates with older data. But there are some cases where even after moving large amounts of data to the cloud tier, the total compression is low. We further analyzed those systems and discovered that those customers are selecting and moving datasets with low total compression to the cloud tier to reduce their on-premises cost. Interestingly, we also found a few systems that have moved 20PB or more to the cloud tier and achieved a total compression factor of 40× or more. One system in particular achieved 66× total compression after moving 40PB of logical data to the cloud tier, resulting in substantial space and cost savings.

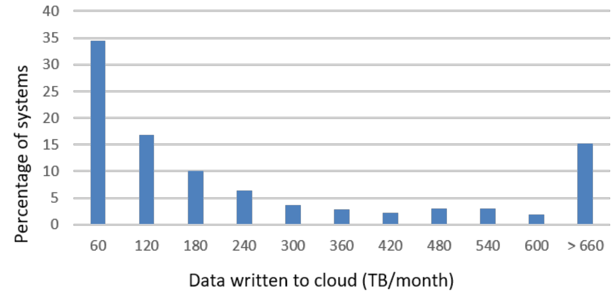


Figure 3: Distribution of TB/month moved to the cloud

Figure 4: Data moved to the cloud versus total compression

To understand the cost savings seen by our customers, we calculated how much money our customers are saving due to deduplication and compression. Even though customers are using various cloud providers, for simplicity, we assume Amazon S3 cost metrics for transaction and storage costs. We calculated total storage costs by calculating a monthly storage cost and accumulating it for all the months. We then calculated the transaction costs based on the total number of transactions performed and the cost of each transaction. Next, we added the storage and transaction costs based on bytes written before and after deduplication and compression, and we calculated the difference of the two. The histogram in Figure 5 shows this difference which represents the cumulative cost savings due to deduplication and compression in thousands of dollars on a log scale. Some of our customers saved nearly $10 million due to deduplication and

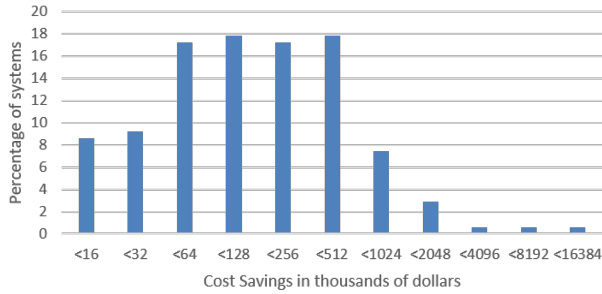


Figure 5: Histogram of cumulative cost savings in thousands of dollars due to deduplication and compression

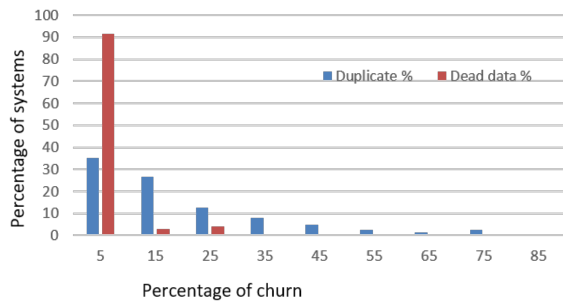


Figure 6: Comparison of physical churn due to file deletes and duplicates

compression. As customers retain the cloud tier data for a longer time period or write more data, their savings due to deduplication and compression will further increase.

8.1.1 Field GC Analysis

To understand how customers are deleting data and how much churn is really generated, we looked at customers who ran GC at-least once (40% of our selected 200 systems). As customers delete files in the cloud tier, data becomes unreferenced and needs to be cleaned. As mentioned in Section 7.2, our system writes some duplicates to improve restore performance. In turn, our GC algorithm retains the most recent version of a duplicate (written to the newest container) and removes older duplicates to reclaim space. Hence there are two types of chunks that can be freed from our system: unreferenced chunks due to file deletions and duplicate chunks. Figure 6 presents the percentage of physical churn (bytes deleted within a time period) that is dead due to file deletions and due to duplicates relative to total capacity of the system. Results from the last three months are included for systems that have run cloud GC at least once.

The churn due to file deletions is relatively low-90% of the systems had less than 5% of space reclaimed for dead chunks, because deduplication creates many references to chunks and because customers tend to retain their cloud data for long periods. There are some cases where the churn due to file deletions is over 25%, suggesting some customers delete a large fraction of their cloud tier periodically. In the steady state, most customers continue to delete the oldest backups as newer backups are stored. This graph also shows that while most of the systems have less than 30% of their churn as duplicates, in some cases the duplicates are very high (up to 75%). We looked at those systems and found that some of those customers have never run GC or have not run GC recently, so duplicates have not yet been removed. In contrast, active tier GC runs each week, so duplicates do not accumulate. We have learned that retention policies differ between active and cloud data, and we have provided an option to customers to control the amount of duplicates they want to write to cloud tier in case they infrequently run GC.

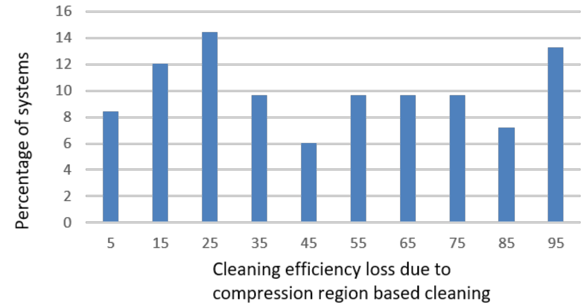


Figure 7: Percentage of cleaning efficiency loss

8.1.2 Cleaning Efficiency Loss due to Compression Region Cleaning

We analyzed the 200 systems to understand how much cleaning efficiency we lose with compression region-based cleaning for the cloud tier relative to possibly running chunk-level cleaning. Figure 7 shows the percentage of bytes that cannot be deleted by compression-region level cleaning but could have been deleted by chunk-level cleaning. As we can see, the cleaning efficiency loss is almost 100% in some cases. In such cases, compression-region level cleaning won't delete anything. We looked at some of these systems closely and found that this happens because the churn (bytes deleted within a time period) in the cloud tier is low. Even though there are a lot of duplicates that can be removed, these duplicates reside in the same compression regions as chunks that are still live, and this prevents us from deleting the compression region. In contrast, chunk-level cleaning is able to delete the duplicates while keeping only the live chunks. Further analysis of this observation is needed as the frequency and pattern of deletions and deduplication can result in different amounts of cleaning efficiency loss. In cases where we are not able to free up any space using compression-region level cleaning, we offer a chunk-level cleaning option, which performs the traditional algorithm of reading and writing container objects over the WAN. We also plan to augment our API and microservice based cleaning services to perform chunk level cleaning in the future.

Hardware	DD-Mid	DD-High
Memory	192GB	384GB
CPU(cores * GHz)	8 * 2.4GHz	24*2.5GHz
Active tier capacity	288TB	720TB
Cloud tier capacity	576TB	1440TB

Table 1: Data Domain hardware for experiments

8.1.3 Deployed Systems Summary

Here are findings from our cloud tier deployments:

1. Some customers are writing 500TB logically per month while others are writing 100TB or less.
2. Customer data is achieving a broad range of total compression ratios, from less than 4× to 100×, because customers are using a cloud tier in different ways. Some are writing highly redundant data to cloud tier (their total compression factor on the cloud and active tiers are both high). Such customers may accumulate more metadata than we originally anticipated and this can affect the runtime of GC and other algorithms. Other customers are writing non-redundant data to the cloud tier. It is likely that such customers are choosing low-deduplication data to migrate to the cloud tier. Other customers have not written much data to the cloud tier yet, so their total compression factor is lower.
3. Some customers have more data written to the cloud tier than to the active tier, so these customers are trying to reduce their on-premises storage cost.
4. Churn on the cloud tier (0-5% per month) is substantially lower than churn on the active tier (10% per week) because customers are not deleting much data. This finding is aligned with our expectation that cloud tier is used for long term retention.
5. Most customers are running cloud tier GC infrequently or not at all and have accumulated a high number of duplicates. Modifications to internal parameters can reduce the number of duplicates in these situations.

8.2 Experiments on Internal Systems

In this section, we focus on results from internal systems experiments. For all experiments, we used two cloud storage systems, Amazon S3 (public cloud) and ECS (private cloud). We used two Data Domain systems shown in Table 1, with the cloud tier feature as described in this paper. The two systems are representative of midrange (DD6800) and high-end (DD9300) products. We provide the size of the active tier and local storage for the cloud tier configuration.

8.2.1 Load Generator

To measure performance of our algorithms, we used an in-house synthetic load generator described previously [1, 11, 7]. A first generation backup is randomly generated, and the following generations have deletions, shuffles, and additions. We typically write many streams in parallel, each stream consisting of generations of backups beginning from a unique seed. The change rate between two consecutive generations is 5%. Gen0 is the first generation of backups where we only get local compression (an average of 2×) and no deduplication. GenX is aged data where generations Gen0-Gen(X-1) are already written to the active or cloud tiers. Due to deduplication, only the new content in GenX is stored while the rest deduplicates. In the experiments below, we vary the initial backup size, number of generations, and number of parallel streams. Beyond 42 generations, the physical locality of data is degraded at a similar level as what many of our customers experience. In one experiment (Section 8.2.4), we generated 100 generations to explore the impact of extremely poor locality.

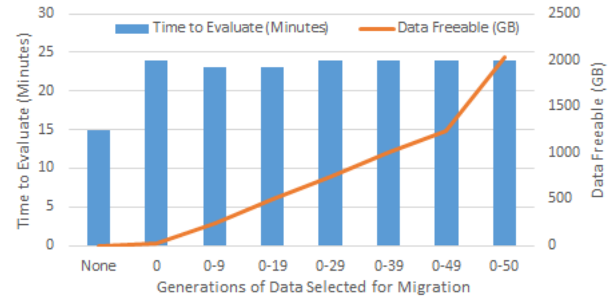


Figure 8: Space estimation performance

8.2.2 Freeable Space Estimation

To evaluate space estimation, we used the synthetic generator to create a data set, selected portions of the data set to potentially migrate, and ran the space estimation algorithm. We created a dataset using 96 parallel streams, each writing generations 0-50 of backups that average 24GB each for a total logical size of 120TB. Figure 8 shows the evaluation time and amount of space that can be freed as we vary the number of generations selected to potentially migrate. With no generations selected (`None`), the evaluation time is 15 minutes, and no space can be freed. As the number of generations selected increases up to including every generation (the rightmost bar), the evaluation time is consistently about 24 minutes while the amount of freeable space increases with the number of generations selected. When all generations are selected, the amount of freeable space jumps since chunks common to the highest generation can finally be freed. These results show that our space estimation algorithm has a run time based on the allocated space of the system. This is because space estimation does physical scans of metadata chunks followed by a container walk to calculate the estimated space that will be freed. The time for both physical scans and a container walk are a function of the physical space on the system.

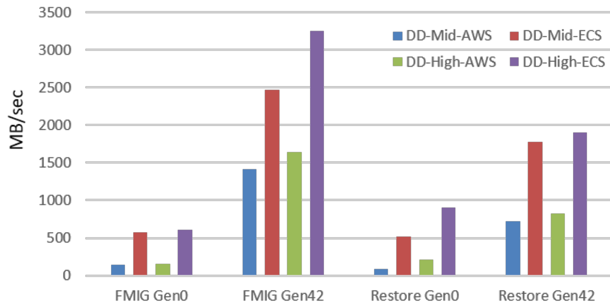


Figure 9: File Migration and Restore performance

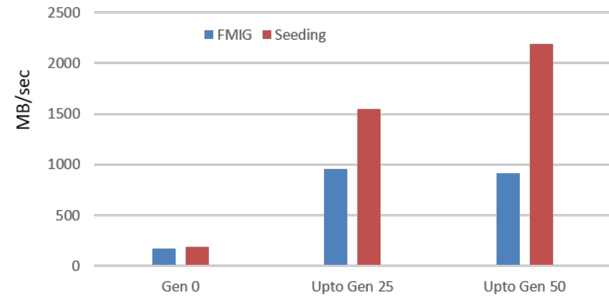


Figure 10: File Migration vs. Seeding performance

The duration of each run is only a few minutes because we wrote a small dataset instead of filling up a 1 PB system, which can take weeks. From experiences with active GC runs on a 1PB system (with the same underlying perfect hashing and physical scans), space estimation on a system of such capacity should finish within a couple of days.

8.2.3 File Migration and Restore from Cloud

Figure 9 shows the logical performance of file migration and restore on DD-Mid and DD-High using both AWS and ECS for object storage. We connected to AWS across the public Internet, while ECS was within our lab. We used Gen42 data to model aged data with high deduplication ratio. Gen42 file migration performance is higher than Gen0 file migration performance because of deduplication, as we only need to transfer the changed data to the cloud tier. In the case of Gen0 file migration, the performance for both hardware configurations is the same. This is because for Gen0, we are reading non-deduplicated data from the active tier and writing to the cloud tier, which is mainly gated by object storage latency and the hardware configuration does not have much impact. But in the case of Gen42, DD-High performs 16% better than DD-Mid on Amazon S3 and 31% better on ECS. This happens because Gen42 has highly deduplicated data compared to Gen0, and DD-High is able to sustain a higher throughput because it has higher parallelism. High object storage latencies continue to be a bottleneck, otherwise the performance difference between the systems would be higher. Gen42 restore from object storage is better than Gen0 restore because in case of Gen42 we are deduplicating against the previous generations of active tier. Also, the latest generation has better locality and hence better performance as GC tries to keep the latest copy of the chunk and hence over time the old generations get fragmented.

Restoring from object storage is typically slower than writing, because it involves reading data from different objects in object storage and writing to the active tier. For comparison, Gen42 write performance on the active tier with the same hardware is 3x better than writing to ECS and 2x better than restore performance from ECS. The major difference in performance is due to object storage latencies. On the active tier latencies of 10-50ms latencies are common, while on the cloud tier these latencies vary from 100 ms to 1 second for both public and private cloud vendors.

8.2.4 File Migration vs. Seeding Performance

Seeding does a bulk transfer of data in a breadth-first manner as opposed to the depth-first manner of file migration. To fairly compare the two algorithms, we did a transfer using both techniques. We ran three sets of tests, where we wrote Gen0, Gen0-50, and Gen0-100 to the active tier and then moved Gen0, Gen0-25, and Gen0-50 to the cloud tier, respectively, using both seeding and file migration. After every 5th generation written, we ran active tier GC and forced it to copy forward 30% of the containers to remove duplicates and degrade the physical locality of the data on the active tier. This simulates the scenario where customers have old, highly deduplicated data on the active tier and would like to move 50% of their oldest data to the cloud tier.

Figure 10 shows that seeding and file migration have similar performance for Gen0 because locality is high and seeding has the overhead of generating the perfect hash functions. As we transfer Gen0-25 and Gen0-50, seeding is faster than file migration by a 2x factor. This is because Gen0-50 have highly deduplicated data with degraded locality and the depth-first approach of file migration has to traverse the same containers repeatedly and incur random I/Os. In the case of seeding, the sequential scans during the breadth-first traversal compensates for the overhead of perfect hashing hence making the movement more efficient. This experiment was only performed on a DD-Mid system with Amazon S3 as the cloud storage. Similar to the Gen0 file migration performance discussed in Section 8.2.3, the hardware configuration does not have much impact because we are bottlenecked by the network throughput.

8.2.5 Garbage Collection Performance

Our analysis focuses on the copy forward process of cloud tier GC, as shown in Figure 11, since it is the only phase that differs from active tier GC. As described in Section 7.2, we developed a new API to perform copy forward for pri-

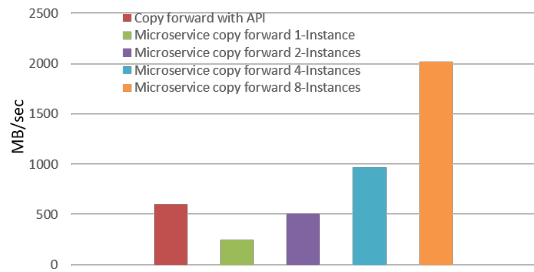


Figure 11: GC copy forward with different algorithms

vate cloud providers. Using this API, the system performs a copy forward of live compression regions (64-128KB) from existing container objects into new container objects without reading the data back to the Data Domain system to avoid transfer costs. Using the API, we are able to achieve a copy forward performance of 600MB/s.

For public cloud providers that do not provide a copy forward API, we developed an experimental microservice algorithm for garbage collection. The Data Domain system makes compression region cleaning decisions and passes a recipe to one or more GC microservice instances running in the cloud that perform the copy forward operations, similar to the functionality of the API. We increased the number of AWS t3.xlarge instances [4] to evaluate how well it scales. Aggregate performance scaled close to linearly, starting at 256 MB/s with one instance and increasing to 2,040 MB/s with eight instances. Even though microservice performance is better than API performance, we can use a microservice only in the case of a public provider where we can spin up compute to run the microservice. This might not be possible in private cloud environments.

9 Related Work

Deduplication is a well-studied field with multiple survey articles [21, 23, 31]. Deduplication is a key aspect of the Data Domain product to enable space savings and high performance for backups [32], and Data Domain has evolved with media and use case changes [1, 2].

There have been multiple papers characterizing backup data [15, 17, 19, 26, 30], and the terms backup and archive are often used interchangeable, so the previous analysis may have applications to archival system design. In our data analysis, customers have specifically decided to archive a subset of their backup data for longer term retention in the cloud.

Reading from deduplicated cloud storage can be slow, and several papers suggest ways to improve read performance, usually involving writing duplicates, caching, and prefetching techniques [18, 27]. Security implications of cloud storage have also been considered [16, 22, 24]. In contrast to these papers, we show how to evolve an existing deduplicated backup product to support a cloud tier.

The issue of deciding where to place large directories to maximize content overlap has been considered [10, 12]. Nagesh et al. [20] presented a technique to partition a collection of files by related content using an in-memory graph relationship on fingerprints. A current publication represents the content of storage volumes with sketches of sampled fingerprints to determine unique content for volumes [14]. In contrast, our technique can estimate the amount of space referenced from an arbitrary set of files selected by the user and scales to PB capacity.

Cumulus [28] provides backups to cloud storage by transferring file differences and storing files in large objects. BlueSky [29] presents a file system backed by cloud storage that uses a local cache for performance. Though neither incorporates deduplication, both projects describe garbage collection for large objects in the cloud as regions become unreferenced. Fu et al. [13] improve GC and restore performance in deduplicated storage by analyzing the history of container references during a backup. They rewrite duplicate chunks from sparse containers from a previous backup and record emerging sparse containers. They also manage container manifests that record which backups reference each container, and when a manifest becomes empty, a container can be safely removed. Such techniques could be used within our cloud GC algorithm, though copy-forward bandwidth is unlikely to be improved.

10 Conclusion

Data protection continues to be a key priority as customers transition their archival data to cloud storage. Data Domain is a mature data protection product, and adding a cloud tier involved trade-offs within the current architecture. We had to make decisions about object sizes and data structure relationships to balance performance and cost not only of migrating data to the cloud tier but also running GC. To address these concerns, we developed several techniques: mirroring metadata locally supports efficient deduplication and GC, and using perfect hashes to track billions of references in memory enables space estimation, seeding, and cloud GC. Experiences with initial customers shows a strong interest in deduplicated archival storage. Large amounts of data are transferred each month, which benefit from deduplication both in terms of faster transfer speeds but also lower storage costs.

Acknowledgments

We thank our shepherd Gala Yadgar and the anonymous reviewers. We thank the many Data Domain filesystem, QA and performance engineers who have contributed to its cloud tier-Bhimsen Bhanjois, Shuang Liang, Kalyani Sundaralingam, Jayasekhar Konduru, Kalyan Gunda, Ashwani Mujoo, Srikant Viswanathan, Chetan Rishud, George Mathew, Prajakta Ayachit, Srikanth Srinivasan, Lan Bai, Abdullah Reza, Kadir Ozdemir, Colin Johnson.

References

[1] ALLU, Y., DOUGLIS, F., KAMAT, M., PRABHAKAR, R., SHILANE, P., AND UGALE, R. Can't We All Get Along? Redesigning Protection Storage for Modern Workloads. In *USENIX Annual Technical Conference (ATC'18)* (2018).

[2] ALLU, Y., DOUGLIS, F., KAMAT, M., SHILANE, P., PATTERSON, H., AND ZHU, B. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer 50*, 7 (2017), 64–72.

[3] AMAZON. Amazon S3. https://aws.amazon.com/s3/, 2018. Retrieved Sept. 17, 2018.

[4] AMAZON. Amazon Web Services. https://aws.amazon.com/, 2018. Retrieved Sept. 17, 2018.

[5] BELAZZOUGUI, D., BOTELHO, F. C., AND DIETZFELBINGER, M. Hash, displace, and compress. In *Algorithms-ESA 2009*. Springer, 2009.

[6] BOTELHO, F. C., PAGH, R., AND ZIVIANI, N. Practical perfect hashing in nearly optimal space. *Information Systems* (June 2012). http://dx.doi.org/10.1016/j.is.2012.06.002.

[7] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

[8] DELL EMC. Elastic Cloud Storage. https://www.dellemc.com/en-us/storage/ecs/, 2018. Retrieved Sept. 17, 2018.

[9] DIETZFELBINGER, M., AND PAGH, R. Succinct data structures for retrieval and approximate membership. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I* (2008), ICALP '08, Springer-Verlag, pp. 385–396.

[10] DOUGLIS, F., BHARDWAJ, D., QIAN, H., AND SHILANE, P. Content-aware load balancing for distributed backup. In *Large Installation System Administration Conference (LISA)* (2011).

[11] DOUGLIS, F., DUGGAL, A., SHILANE, P., WONG, T., YAN, S., AND BOTELHO, F. C. The logic of physical garbage collection in deduplicating storage. In *USENIX Conference on File and Storage Technologies (FAST'17)* (2017).

[12] FORMAN, G., ESHGHI, K., AND SUERMONDT, J. Efficient detection of large-scale redundancy in enterprise file systems. *SIGOPS Oper. Syst. Rev. 43*, 1 (Jan. 2009), 84–91.

[13] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[14] HARNIK, D., HERSHCOVITCH, M., SHATSKY, Y., EPSTEIN, A., AND KAT, R. Sketching volume capacities in deduplicated storage. In *USENIX Conference on File and Storage Technologies (FAST'19)* (2019).

[15] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of The Israeli Experimental Systems Conference* (2009), ACM.

[16] LI, J., LI, Y. K., CHEN, X., LEE, P. P., AND LOU, W. A hybrid cloud approach for secure authorized deduplication. *IEEE Transactions on Parallel and Distributed Systems 26*, 5 (2015), 1206–1216.

[17] LU, M., CHAMBLISS, D., GLIDER, J., AND CONSTANTINESCU, C. Insights for data reduction in primary storage: a practical analysis. In *Proceedings of the International Systems and Storage Conference* (2012), ACM.

[18] MAO, B., JIANG, H., WU, S., FU, Y., AND TIAN, L. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Transactions on Storage 10*, 2 (Mar. 2014).

[19] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *USENIX Conference on File and Storage Technologies (FAST'11)* (2011).

[20] NAGESH, P., AND KATHPAL, A. Rangoli: Space management in deduplication environments. In *Proceedings of the International Systems and Storage Conference* (2013), ACM.

[21] NEELAVENI, P., AND VIJAYALAKSHMI, M. A survey on deduplication in cloud storage. *Asian Journal of Information Technology 13*, 6 (2014), 320–330.

[22] NG, W. K., WEN, Y., AND ZHU, H. Private data deduplication protocols in cloud storage. In *Proceedings of the ACM Symposium on Applied Computing* (2012), ACM, pp. 441–446.

[23] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[24] Puzio, P., Molva, R., Onen, M., and Loureiro, S. Cloudedup: secure deduplication with encrypted data for cloud storage. In *International Conference on Cloud Computing Technology and Science (CloudCom)* (2013), vol. 1, IEEE, pp. 363–370.

[25] Srinivasan, K., Bisson, T., Goodson, G. R., and Voruganti, K. iDedup: latency-aware, inline data deduplication for primary storage. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[26] Sun, Z. J., Kuenning, G., Mandal, S., Shilane, P., Tarasov, V., Xiao, N., and Zadok, E. Cluster and single-node analysis of long-term deduplication patterns. *ACM Trans. Storage 14*, 2 (May 2018), 13:1–13:27.

[27] Tan, Y., Jiang, H., Feng, D., Tian, L., and Yan, Z. Cabdedupe: A causality-based deduplication performance booster for cloud backup services. In *IEEE International Conference on Parallel & Distributed Processing Symposium (IPDPS)* (2011).

[28] Vrable, M., Savage, S., and Voelker, G. M. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS) 5*, 4 (2009), 14.

[29] Vrable, M., Savage, S., and Voelker, G. M. Bluesky: A cloud-backed file system for the enterprise. In *USENIX conference on File and Storage Technologies (FAST'12)* (2012).

[30] Wallace, G., Douglis, F., Qian, H., Shilane, P., Smaldone, S., Chamness, M., and Hsu, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[31] Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y., and Zhou, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016).

[32] Zhu, B., Li, K., and Patterson, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (2008).

GAIA: An OS Page Cache for Heterogeneous Systems

Tanya Brokhman, Pavel Lifshits and Mark Silberstein
Technion – Israel Institute of Technology

Abstract

We propose a principled approach to integrating GPU memory with an OS page cache. We design *GAIA*, a weakly-consistent page cache that spans CPU and GPU memories. GAIA enables the standard `mmap` system call to map files into the GPU address space, thereby enabling data-dependent GPU accesses to large files and efficient write-sharing between the CPU and GPUs. Under the hood, GAIA (1) integrates lazy release consistency protocol into the OS page cache while maintaining backward compatibility with CPU processes and *unmodified* GPU kernels; (2) improves CPU I/O performance by using data cached in GPU memory, and (3) optimizes the readahead prefetcher to support accesses to files cached in GPUs.

We prototype GAIA in Linux and evaluate it on NVIDIA Pascal GPUs. We show up to 3× speedup in *CPU* file I/O and up to 8× in *unmodified* realistic workloads such as Gunrock GPU-accelerated graph processing, image collage, and microscopy image stitching.

Introduction

GPUs have come a long way from fixed-function accelerators to fully-programmable, high-performance processors. Yet their integration with the host Operating System (OS) is still quite limited. In particular, GPU physical memory, which today may be as large as 32GB [9], has been traditionally managed entirely by the GPU driver, without the host OS control. One crucial implication of this design is that the OS cannot provide core system services to GPU kernels, such as efficient access to memory mapped files, nor can it optimize I/O performance for CPU applications sharing files with GPUs. To mitigate these limitations, tighter *integration of GPU memory into the OS page cache and file I/O mechanisms* is required. Achieving such integration is one of the goals of this paper.

Prior works demonstrate that mapping files into GPU memory provides a number of benefits [34, 36, 35]. Files can be accessed from the GPU using an intuitive pointer-based programming model, enabling GPU applications with data-dependent access patterns. Transparent system-level performance optimizations such as prefetching and double buffering can be implemented to achieve high performance for I/O intensive GPU kernels. Finally, file contents can be easily shared between legacy CPU and GPU-accelerated processes.

Extending the OS page cache into GPU memory is advantageous even for CPU I/O performance. With modern servers commonly hosting 8 and more GPUs, the total GPU memory available (100-200GB) is large enough to be used for caching file contents. As we show empirically, doing so may boost the I/O performance by up to 3× compared to accesses to a high-performance SSD (§6). Finally, the OS management of the page cache in GPU memory may allow caching GPU file accesses directly in the GPU page cache, bypassing CPU-side page cache and avoiding its pollution [15].

Unfortunately, today's commodity systems fall short of providing full integration of GPU memory with the OS page cache. ActivePointers [34] enable a memory-mapped files abstraction for GPUs, but their use of special pointers requires intrusive modifications to GPU kernels, making them incompatible with closed-source libraries such as cuBLAS [7]. NVIDIA's Unified Virtual Memory (UVM) [8] and the Heterogeneous Memory Management (HMM) [4] module in Linux allow GPUs and CPUs to access shared virtual memory space. However, neither UVM nor HMM allow mapping files into GPU memory, which makes them inefficient when processing large files (§6.3.2). More fundamentally, both UVM and HMM force the physical page to be present in the memory of only one processor. This results in a performance penalty in case of false sharing in data-parallel write-sharing workloads. Moreover, false sharing has a significant impact on the system as a whole, as we show in (§3).

Several hardware architectures introduce cache coherence between CPUs and GPUs. In particular, CPUs with integrated GPUs support coherent shared *virtual* memory in hardware. In contrast to discrete GPUs, however, integrated GPUs lack large separate physical memory. Therefore, today's OSes do not provide any memory management services for them.

Recent high-end IBM Power-9 systems feature hardware cache-coherent shared virtual memory between CPUs and discrete GPUs [31]. GPU memory is managed as another NUMA node. Thus, the OS is able to provide memory management services to GPUs, including memory-mapped files. Unfortunately, cache coherence between the CPU and discrete GPUs is not available in x86-based commodity systems, and it is unclear when it will be introduced (see §4.3). Clearly, using the NUMA mechanisms for non-coherent GPU memory

management would not work. For example, migrating a CPU-accessible page into GPU memory will break the expected memory behavior for CPU processes, e.g., due to the lack of atomic operations across the PCIe bus, among other issues.

To resolve these limitations, we propose **GAIA**[1], a distributed, weakly-consistent page cache architecture for heterogeneous multi-GPU systems that extends the OS page cache into GPU memories and integrates with the OS file I/O layer. With GAIA, CPU programs use regular file I/O to share files with GPUs. Calling `mmap` with a new `MMAP_ONGPU` flag makes the mapping accessible to the GPU kernels, thus providing support for GPU accesses to shared files. This approach allows access to memory-mapped files from *unmodified* GPU kernels.

This paper makes the following contributions:

- We characterize the overheads of CPU-GPU false sharing in existing systems (§3.1). We propose a unified page cache which eliminates false sharing by using a lazy release consistency model [22, 10].
- We extend the OS page cache to control the *unified* (CPU and GPU) page cache and its consistency (§4.1), without requiring CPU-GPU hardware cache coherence. We introduce a *peer-caching* mechanism and integrate it with the OS readahead prefetcher, enabling any processor accessing files to retrieve them from the best location, and in particular, from GPU memory (§6.2).
- We present a fully functional generic implementation in Linux, *not tailored* to any particular GPU.
- We prototype GAIA on NVIDIA Pascal GPU, leveraging its page fault support. We modify public parts of the GPU driver and reliably emulate the functionality which cannot be implemented due to the closed-source driver.
- We evaluate GAIA using real workloads, including (1) an *unmodified* graph processing framework - Gunrock [38], (2) a Mosaic application that creates an image collage from a large image database [34], and (3) a multi-GPU image stitching application [16, 18] that determines the optimal way to combine multiple image tiles, demonstrating the advantages and the ease-of-use of GAIA for real-life scenarios.

Background

We briefly explain the main principles of several existing memory consistency models relevant to our work.

Release consistency. Release consistency (RC) [22] is a form of relaxed memory consistency that permits delaying the effects of writes to distributed shared memory. The programmer controls the visibility of the writes from each processor by means of the *acquire* and *release* synchronization operations. Informally, the writes are guaranteed to be visible to the readers of a shared memory region after the writer *release*-s the region and the reader *acquire*-s it.

RC permits concurrent updates to different versions of the page in multiple processors, which get *merged* upon later accesses. A common way to resolve merge conflicts is by using the version vectors mechanism, explained below.

Lazy release consistency. In Lazy Release Consistency (LRC) [22, 10] the propagation of updates to a page is delayed until *acquire*. At synchronization time, the acquiring processor receives the updates from the other processors. Usually, the underlying implementation leverages page faults to trigger the updates [22]. Specifically, a stale local copy of the page is marked inaccessible, causing the processor to fault on the first access. The faulting processor then retrieves the up-to-date copy of the page from one or more processors. In the home-based version of the protocol, a home node is assigned to a page to maintain the most up-to-date version of the page. The requesting processor contacts the home node to retrieve the latest version of the page.

Version vectors. Version vectors (VVs) [32] are used in distributed systems to keep track of replica versions of an object. The description below is transcribed from Parker et al. [32].

A version vector of an object O is a sequence of n pairs, where n is the number of sites at which O is stored. The pair $\{S_i : v_i\}$ is the latest version of O made at site S_i. That is, the vector entry v_i counts the number of updates to O made at site S_i. Each time O is copied from site S_i to S_j at which it was not present, the version vector of site S_i is adopted by site S_j. If the sites have a conflicting version of the replica, the new vector is created by taking the largest version among the two for each entry in the vector.

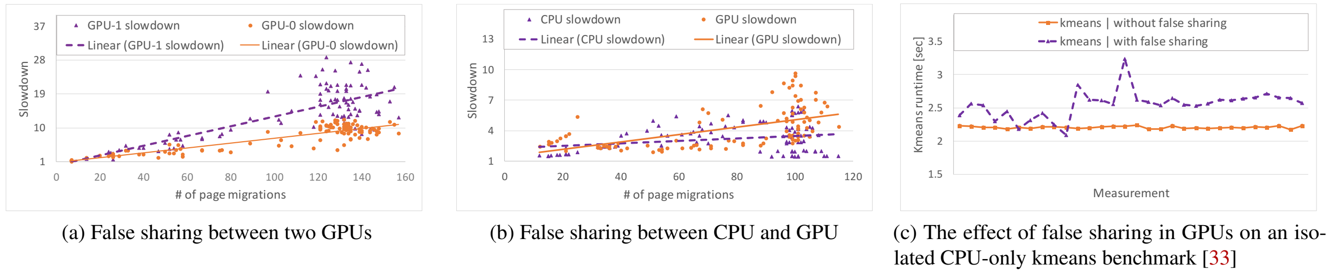
Consistency model considerations

The choice of the consistency model for the unified page cache is an important design question. We describe the options we considered and justify our choice of LRC.

POSIX: strong consistency. In POSIX writes are immediately visible to all the processes using the same file [5]. In x86 systems without coherent shared memory and with GPUs connected via a high latency (relative to local memory) PCIe bus, such a strong consistency model in a unified page cache would be inefficient [36].

GPUfs: session semantics. GPUfs [36] introduces a GPU-side library for file I/O from GPU kernels. GPUfs implements a distributed page cache with session semantics. Session semantics, however, couple between the file open/close operations and data synchronization. As a result, they cannot be used with `mmap`, as sometimes the file contents need to be synchronized across processors without having to unmap and close the file and then reopen and map it again. Therefore, we find session semantics unsuitable for GAIA.

[1] **G**lob**A**l un**I**fied p**A**ge cache, or simply the daughter's name of one of the authors, born during this project.



(a) False sharing between two GPUs

(b) False sharing between CPU and GPU

(c) The effect of false sharing in GPUs on an isolated CPU-only kmeans benchmark [33]

Figure 1: Impact of false sharing on the performance of GPU kernels and the system as a whole.

UVM: page-level strict coherence. NVIDIA UVM [8] and Linux (HMM) [4] implement strict coherence [24] at the GPU page granularity (e.g., 64KB in NVIDIA GPUs). In this model, a page can be mapped only by one processor at a time. Thus, two processors cannot observe different replicas of the page (multiple read-only replicas are allowed). If a processor accesses a non-resident page, the page fault causes the page to migrate, i.e., the data is transferred, and the page is remapped at the requestor and unmapped at the source.

Although this model might seem appealing for page cache management, it suffers from sporadic performance degradation due to *false sharing*. False sharing of a page occurs when two processors inadvertently share the same page, at least one of them for write, while performing non-overlapping data accesses [17]. False sharing is known to dramatically degrade the performance of distributed shared memory systems with strict coherence because it causes repetitive and costly page migration among different physical locations [17]. False sharing has been also reported in multi-GPU applications that use NVIDIA's UVM [8]. The official recommended solution is to allocate private replicas of the shared buffer in each processor and manually merge them after use.

False sharing in a page cache. If strict coherence is used for managing a unified page cache, false sharing of the page cache pages might occur quite often. Consider an image processing task that stitches multiple image tiles into a large output image stored in a file, e.g., when processing samples from a microscope [16]. False sharing will likely occur when multiple GPUs process the images in a data-parallel way, each writing its results to a shared output file. Consider two GPUs, one processing the left and another the right half of the image. In this case, false sharing might occur at every *row* of the output. This is because for large images (thousands of pixels in each dimension) stored in row-major format, each row will occupy at least one memory page in the page cache. Since each half of the row is processed on a different GPU, the same page will be updated by both GPUs. We observe this effect in real applications (§ 6.3.3).

False sharing with UVM

Impact of false sharing on application performance. To experimentally quantify the cost of false sharing in multi-GPU systems, we allocate a 64KB-buffer (one GPU page) and divide it between two NVIDIA GTX1080 GPUs. Each GPU executes read-modify-write operations (so they are not optimized out) on its half in a loop. We run a total of 64 threadblocks per GPU, each accessing its own part of the array, all active during the run. To control the degree of contention, we vary the number of loop iterations per GPU.

We compare the execution time when both GPUs use a shared UVM buffer (with false sharing) with the case when both use private buffers and merge them at the end of the run (no false sharing). Figure 1a shows the scatter graph of the measurements. False sharing causes slowdown that grows with the number of page migrations, reaching 28×, and results in large runtime variance [2]. Figure 1b shows similar results when one of the GPUs is replaced with a single CPU thread. This also indicates that adding more GPUs is likely to cause even larger degradation due to higher contention and increased data transfer costs.

System impact of false sharing. False sharing among GPUs affects the performance of the system as a whole. We run the CPU-only kmeans benchmark from Phoenix [33] *in parallel* with the multi-GPU false sharing benchmark above. We allocate two CPU cores for GPU management, and the remaining four cores to running kmeans (modified to spawn four threads). The GPU activity *should not* interfere with kmeans because kmeans does not use the GPU.

However, *we observe significant interference when GPUs experience false sharing*. Figure 1c depicts the runtime of kmeans when run together with the multi-GPU run, with and without false sharing. Not only does kmeans become up to 47% slower, but the execution times vary substantially. Thus, false sharing affects an *unrelated* CPU application, breaking the fundamental OS performance isolation properties.

Preventing page bouncing via pinning. In theory, the false sharing overheads could be reduced by pinning the page in

[2]The difference in the slowdown between the two GPUs stems from the imperfect synchronization between them. Thus, the one invoked first (GPU0) can run briefly without contention.

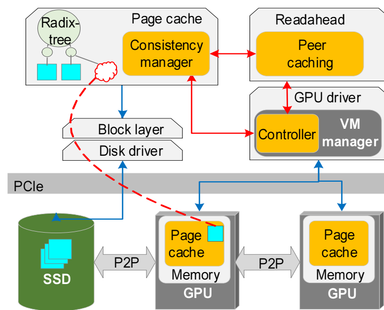


Figure 2: GAIA high-level design in the OS kernel. The modified parts are highlighted.

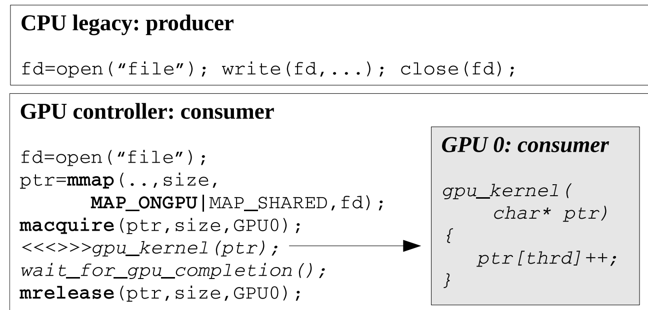


Figure 3: Code sketch of `mmap` for GPU. The CPU writes data into the file and then invokes the GPU controller, which maps the file and runs the GPU kernel.

memory of one of the processors, and mapping it into the address space of the other processors for remote access over PCIe. Unfortunately, pinning page cache pages is quite problematic. It would require substantial modifications to the existing page cache management mechanisms. For example, to be evicted, the pinned page would need to be unmapped from the virtual address space of all the mapping processors.

Moreover, even though pinning is likely to yield better system performance for pages experiencing false sharing, in the common case remote accesses from other processors would be slower than accessing the pages locally. Thus, robust false sharing detection heuristics should be designed, such that only the actual page bouncing triggers the pinning mechanism. On the other hand, enabling the programmer to pin pages manually at the `mmap` time is not efficient either, because then the pages must be initialized with the contents of the file. Mapping large files would thus require reading them in full from the disk, which not only nullifies the on-demand file loading benefits of `mmap`, but might not even be possible for the large files exceeding physical memory.

Implications for Unified Page Cache design. We conclude that *the UVM strict coherence model is unsuitable for implementing a unified page cache*. It may suffer from spurious and hard-to-debug performance degradation that affects the whole system, and only worsens as the number of GPUs increases. A system-level service with such inherent limitations would be a poor design choice. Thus, we chose to build a unified cache that follows the lazy-release consistency model and sidesteps the false sharing issues entirely.

Design

Overview. Figure 2 shows the main GAIA components in the OS kernel. A distributed page cache spans across the CPU and GPU memories. The OS page cache is extended to include a *consistency manager* that implements home-based lazy release consistency (LRC). It keeps track of the versions of all the file-backed memory pages and their locations. When a page is requested by the GPU or the CPU (due to a page fault), the consistency manager determines the locations of the most recent versions, and retrieves and merges them if necessary. We introduce new `macquire` and `mrelease` system calls which follow standard Release Consistency semantics and have to be used when accessing shared files. We explain the page cache design in (§4.1).

If an up-to-date page replica is available in multiple locations, the *peer-caching* mechanism retrieves the page via the most efficient path, e.g., from GPU memory for the CPU I/O request, or directly from storage for the GPU access as in SPIN [15]. This mechanism is integrated with the OS readahead prefetcher to achieve high performance (§6.2). To enable proper handling of memory-mapped files on GPUs, the GAIA controller in the GPU driver keeps track of all the GPU virtual ranges in the system that are backed by files.

File-sharing example. Figure 3 shows a code sketch of sharing a file between a legacy CPU application (producer) and a GPU-accelerated one (consumer). This example illustrates two important aspects of the design. First, *no GPU kernel changes* are necessary to access files, and no new system code runs on the GPU. The consistency management is performed by the CPU consumer process that uses the GPU, which we call the *GPU controller*. Second, *no modifications to legacy CPU programs* are required to share files with GPUs or among themselves, despite the weak page cache consistency model. The consistency control logic is confined to the GPU controller process. Besides being backward compatible, this design simplifies integration with the CPU file I/O stack.

Consistency manager

Version vectors. GAIA maintains the version of each file-backed 4K page for every entity that might hold the copy of the page. We call such an entity a *page owner*. We use the well-known version vector mechanism (§2) to allow scalable version tracking for each page [32].

A page owner might be a CPU, each one of the GPUs, or the storage device. Keeping track of the storage copy is important because GAIA supports direct transfer from the disk to GPU memory. Consider the example in Figure 4, where a page is first concurrently modified by the CPU and the GPU, and then flushed to storage by the CPU. Flushing it from the CPU

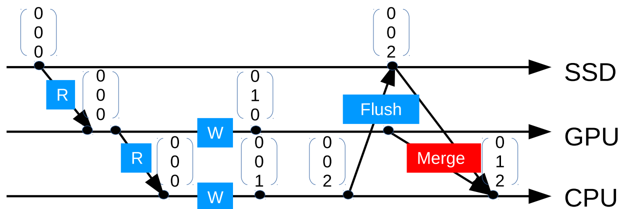


Figure 4: Version vectors in GAIA. The CPU flushes its replica to disk, the GPU keeps its version. The following CPU read must merge two replicas.

removes both the data and its version information from CPU memory. The next reader must be able to retrieve the most recent version of the page, which in our example requires *merging* the page versions on the disk and on the GPU. The storage entry in the version vector is always zero.

A new *Time Stamp Version Table (TSVT)* stores all the version vectors for a page. This table is located in the respective node of the page cache radix tree. The GPU entries are updated by the CPU-side GPU controller on behalf of the GPU. We choose the CPU-centric design to avoid intrusive modifications to GPU software and hardware.

Synchronizing system calls for consistency control. We introduce two new system calls to implement LRC.

`macquire(void *addr, size len, void* device)` must be called to ensure that the `device` accesses the latest version of the data in the specified address range. `macquire` scans through the address range on the `device` and invalidates (unmaps and drops) all the outdated local pages. When called for the CPU, it unmaps such pages from all the CPU processes. Thus, the following access to the page will cause a page fault trap to retrieve the most recent version of the page, as we describe later in (§4.1.1).

`mrelease(void *addr, size_t len, void* device)` must be called by the `device` that writes to the respective range to propagate the updates to the rest of the system. Similarly to `macquire`, this operation does not involve data movements. It only increases the versions of all the modified (since the last `macquire`) pages in the owner's entry of its version vector.

Tracking the status of CPU pages requires a separate `LRC_modified` flag in the page cache node, in addition to the original modified flag used by the OS page cache. This is because the latter can be reset by other OS mechanisms, e.g, flush, resulting in a missed update. The new flag is set together with the original one, but is reset by `mrelease` call as part of the version vector update.

Transparent consistency support for the CPU. GAIA does not change the original POSIX semantics when sharing files among CPU processes, because all the CPUs share the same replica of the cached page. However, `macquire` and `mrelease` calls must be invoked by *all* the CPU processes that might inadvertently share files with GPUs. In GAIA we seek to eliminate this requirement.

Our solution is to perform the CPU synchronization calls eagerly, combining them with `macquire` and `mrelease` calls issued on behalf of GPUs. The `macquire` call for the GPU is invoked after internally calling `mrelease` for the CPU, and `mrelease` of the GPU is always followed by `macquire` for the CPU. This change does not affect the correctness of the original LRC protocol, because it maintains the relative ordering of the acquire and release calls on different processors, simply moving them closer to each other.

Consistency and GPU kernel execution. GAIA's design does not preclude invocation of `macquire` and `mrelease` during the kernel execution on the target GPU. However, the current prototype does not support such functionality, because we cannot retrieve the list of dirty pages from the GPU while it is running, which is necessary for implementing `mrelease`. Therefore, we support the most natural scenario (also in Figure 3), which is to invoke `macquire` and `mrelease` at the kernel execution boundaries. Integrating these calls with the CUDA streaming API might be possible by using CUDA CPU callbacks [2]. We leave this for future work.

Page faults and merge

Page faults from any processor are handled by the CPU (hence, home-based LRC). CPU and GPU-originated page faults are handled similarly. For the latter, the data is moved to the GPU. The handler locates the latest versions of the page according to its TSVT in the page cache. If the faulting processor holds the latest version in its own memory (minor page fault), the page is remapped. If, however, the present page is outdated or not available, the page is retrieved from the memory of other processors or from storage.

This process involves handling the merge of multiple replicas of the same page. The overlapping writes to the same memory locations (i.e., the actual data races) are resolved via an "any write wins" policy, in a deterministic order (i.e., based on the device hardware ID). However, non-overlapping writes to the same page must be explicitly merged via 3-way merge, as in other LRC implementations [22].

3-way merge. The CPU creates a *pristine* base copy of the page when a GPU maps the page as writable. Conflicting pages are compared with their base copies first to detect the changes.

The storage overheads due to pristine copies might compromise scalability, as we discuss in (§6.1). The naive solution is to store a per-GPU copy of each page. A more space-efficient design might use a single page base copy for all the processors, employing copy-on-write and eagerly propagating the updates after the processor `mrelease`-s the page, instead of waiting for the next page fault (lazy update). Our current implementation uses the simple variant.

The overheads of maintaining the base copy are not large in practice. First, the base copy can be discarded after the page is evicted from GPU memory. Further, it is not required for

read-only accesses, or when there is only one-page owner (excluding the storage) in the system. Most importantly, creating the base copy is not necessary for writes from CPU processes. This is because the CPU is either the sole owner, or the base copy has already been created for the modifying GPU.

Interaction with file I/O

Peer-caching. GAIA architecture allows a page replica to be cached in multiple locations, so that the best possible I/O path (or possibly multiple I/O paths in parallel) can be chosen, to serve page access requests. In particular, the CPU I/O request can be served from GPU memory. Note that access to the GPU-cached page does not invalidate it for the GPU.

A naive approach to peer-caching is to determine the best location individually for each page. However, this approach *degrades* the performance for sequential accesses by an order of magnitude, due to the overheads of small data transfers over the PCIe bus. Instead, GAIA leverages the OS prefetcher to optimize PCIe transfers. We modify the prefetcher to determine the data location in conjunction with deciding how much data to read at once. This modification results in a substantial performance boost, as we show in (§6.2).

Readahead for GPU streaming access. GPUs may concurrently run hundreds of thousands of threads that access large amounts of memory at once. GPU hardware coalesces multiple page faults together (up to 256, one for 64KB page). If the page faults are triggered by accesses to the memory-mapped file on the GPU, GAIA reads the file according to the GPU-requested locations and copies the data to GPU pages. We call such accesses *GPU I/O*.

We observe that the existing OS readahead prefetcher does not work well for GPU I/O. It is often unable to optimize streaming access patterns where a GPU kernel reads the whole file in data-parallel strides, one stride per group of GPU threads. The file accesses from the GPU in such a case appear random when delivered to the CPU due to the non-deterministic hardware schedule of GPU threads, thereby confusing the CPU read-ahead heuristics.

We modify the existing OS prefetcher by adjusting the upper bound on the read-ahead window to 16MB (64× of the CPU), but only for GPU I/O accesses. We also add `madvise` hints that increase the minimum read size from a disk to 512KB for sequential accesses. These changes allow the prefetcher to retrieve more data faster when the sequential pattern is recognized, but it does not fully recover the performance. Investigating a better prefetcher heuristic that can cope with massive multi-threading is left for future work.

Discussion

GAIA and cache-coherent accelerator architectures. Cache-coherent systems with global virtual address space may allow a simpler solution to the page cache management. However, we believe that cache coherence between the CPU and discrete accelerators is unlikely to fully replace existing systems soon. Despite the cache coherent technologies (CAPI [37]) having been available, the high cost and the need for industry-wide coordination on open interfaces have hindered their adoption thus far. Nor is it apparent how and to what extent these technologies will improve commodity applications (i.e., graphics, deep learning). Many additional open issues (for example, scalability) also must be addressed. Therefore, in GAIA we choose not to rely on cache-coherence among CPUs and accelerators.

No snapshot isolation. GAIA does not provide snapshot isolation. This is consistent with prior work on GPU file system support [36]. While adding such guarantee is possible, we did not find applications that require it.

Prefetching hints. Our current prototype could be extended to support more advanced prefetching hints similar to UVM [1]. For example, it could employ eager data copy into the page cache of a specific GPU that is known to exclusively access the data. We leave this for future work.

Using huge CPU pages. GAIA design and implementation is tailored for 4KB pages managed by the OS. However, GAIA can be adapted to support different page sizes as well, i.e. 2MB huge pages. Huge pages require only minor modifications to the TSVT management logic and tables, and might improve performance for applications with sequential file access. This is because transferring 2MB pages over PCIe is about 5× more efficient than 4KB pages. On the other hand, increasing the page sizes would affect the workloads with poor spatial locality, such as Mosaic (§6.3.1).

GAIA compatibility with other accelerators. GAIA's design can be extended to other GPUs and accelerators with paging capabilities. In fact, support for paging was introduced recently in AMD GPUs [4, 6]. However, implementation in GAIA would require an accelerator to expose minimal page management APIs, as we explain in the next section.

Implementation

GAIA implementation requires changing 3300 LOC and 1200 LOC in Linux kernel and the NVIDIA UVM driver respectively.

OS changes

Page cache with GPU pointers. In Linux, the page cache is represented as a per-file radix tree with each leaf node corresponding to a continuous 256KB file segment. Each leaf holds an array of addresses (or NULLs) of 64 physical pages caching the file content.

GAIA extends the tree leaf node data structure to store the addresses of GPU pages. We add 4 pointers per leaf node per GPU, to cover a continuous 256KB file segment (GPU page is

	Function	Purpose	UVM implementation\ GAIA emulation	Used by
Available in UVM	allocVirtual/allocPhysical mapP2V	allocate virtual/physical range map physical-to-virtual	cudaMallocManaged()	mmap
	freeVirtual/freePhysical	free virtual/physical range unmap virtual	cudaFree()	munmap
Emulated by GAIA	unmapV	Invalidate mapping in GPU	Migrate page to CPU	maquire
	fetchPageModifiedBit	Retrieve dirty bit in GPU	Copy page to CPU and compute diff	mrelease

Table 1: Main GPU Virtual Memory management functions and their implementation with UVM

64KB). The CPU only keeps track of file-backed GPU pages rather than the entire GPU physical memory. The leaf node stores all the versions (TSVT) for the 64 4KB pages.

Linking the page cache with the GPU page ranges. GAIA keeps track of all the GPU virtual ranges in the system that are backed by files to properly handle the GPU faults for file-backed pages. When `mmap` allocates a virtual address range in the GPU via the driver, it registers the range with GAIA and associates it with the file radix tree.

Data persistence. GAIA inherits the persistence semantics of the Linux file I/O. It updates both `msync` and `fsync` to fetch the fresh versions of the cache pages (similarly to the logic in the page fault handler) and write their contents to storage.

GPU cache size limit. GAIA enforces an upper bound on the GPU cache size by evicting pages. The evicted pages can be discarded from the system memory entirely (after syncing with the disk if necessary) or cached by moving them to available memory of other processors. In our current implementation, we cache the evicted GPU pages in CPU memory. We implement the Least Recently Allocated eviction policy [36], due to the lack of the access statistics for GPU pages.

Integration with GPU driver

The NVIDIA GPU driver provides no public interface for low-level virtual memory management. Indeed, giving the OS the full control over GPU memory management might seem undesirable. For example, only the vendors might have the intimate knowledge of the device/vendor-specific properties that require special handling, such as different page sizes, texture memory, alignment requirements, and physical memory constraints. However, we believe that a minimal subset of APIs is enough to allow generic advanced OS services for GPUs, such as unified page cache management, without forcing the vendors to give up on the GPU memory control.

We define such APIs in Table 1. The driver is in full control of the GPU memory, i.e., it performs allocations and implements the page eviction policy, only notifying the OS about the changes (callbacks are not shown in the table). We demonstrate the utility of such APIs for GAIA, encouraging GPU vendors to add them in the future.

Using the GPU VM management API

Implementing GAIA functionality is fairly straightforward, and closely follows the implementation of the similar functionality in the CPU OS. We provide a brief sketch below just to illustrate the use of the API.

mmap/munmap. When `mmap` with `MAP_ONGPU` is called, the system allocates a new virtual memory range in GPU memory via `allocVirtual` and registers it with the GAIA controller in the driver to associate it with the respective file radix tree, similar to the CPU `mmap` implementation. The `munmap` function performs the reverse operation using `unmapV` call.

Page fault handling. To serve the GPU page fault, GAIA determines the file offset and searches for the pages that cache the content in the associated page cache radix tree. If the most recent version of the page is found, and it is already located on the requesting GPU (minor page fault), GAIA maps the page at the appropriate virtual address using `mapP2V` call.

Otherwise (major page fault), GAIA allocates a new GPU physical page via `allocPhysical` call, populates it with the appropriate data (merging replicas if needed), updates the page's TSVT structure in the page cache to reflect the version, and maps the page to the GPU virtual memory. If necessary, GAIA creates a pristine copy of the page.

If a page has to be evicted to free space in GPU memory, GAIA chooses the victim page, unmaps it via `unmapV`, retrieves its dirty status via `fetchPageModifiedBit`, stores the page content on the disk or CPU memory if marked as dirty, removes the page reference from the page cache, and finally frees it via `freePhysical`.

Consistency system calls. GPU `macquire` scans through the GPU-resident pages of the page cache to identify outdated GPU page replicas and unmaps the respective pages via `unmapV`. GPU `mrelease` retrieves the modified status via `fetchPageModifiedBit` for all the GPU-resident pages in the page cache, and updates their versions in TSVT.

Functional emulation of the API

Implementing the proposed GPU memory management API without vendor support requires access to low-level internals of the closed-source GPU driver. Therefore, we choose to implement it in a limited form.

First, we use the user-level NVIDIA's UVM memory management APIs to implement a limited version of the API for allocation and mapping physical and virtual pages in GPU (refer to Table 1). Specifically, `cudaMallocManaged` is used to allocate the GPU virtual address range, and `cudaFree` to tear down the mapping and de-allocate memory.

Second, we modify the open-source part of the NVIDIA UVM driver. The GPU physical page allocation and mapping of the virtual to physical addresses are all part of the GPU page fault handling mechanism, yet they are implemented in the closed-source part of the UVM driver. To use them, GAIA modifies the open-source UVM page fault handler to perform the file I/O and page cache-related operations, effectively implementing the scheme described above (§5.2.1).

Finally, whenever the public APIs and open-source part of the driver are insufficient, we resort to emulation. To implement `unmapV`, we use a public driver function to *migrate the page* to the CPU, which also unmaps the GPU page. The `fetchPageModifiedBit` call is emulated by copying the respective page to the CPU *without unmapping it on the GPU* and computing *diff* with the base copy.

In Table 1 we highlight the emulated functions (in red) and specify where they are used.

Ultimately, this pragmatic approach allows us to build a functional prototype to evaluate the concepts presented in the paper. We hope that tese APIs will be implemented properly by GPU vendors in the future.

Limitations due to UVM

Our forced reliance on NVIDIA UVM leads to several limitations. The page cache lifetime and scope are limited to those of the process where the mapping is created, as UVM does not allow allocating physical pages that do not belong to a GPU context. Therefore, the page cache cannot be shared among GPU kernels belonging to different CPU processes. Further, the maximum mapped file size is limited by the size of the maximum UVM buffer allocation, which must fit in the CPU physical memory. Finally, UVM controls memories of all the system GPUs, *preventing us from implementing a distributed page cache between multiple NVIDIA GPUs*.

These limitations are rooted in our use of UVM, and *are not pertinent to GAIA design*. They limit the scope of our evaluation to a single CPU process and a single GPU, but allow us to implement a substantial prototype to perform thorough and reliable performance analysis of the heterogeneous page cache architecture.

Evaluation

We evaluate the following aspects of our system [3]:

- Benefits of peer-caching and prefetching optimizations;
- Memory and compute overheads;
- End-to-end performance in real-life applications with read and write-sharing.

Platform. We use an Intel Xeon CPU E5-2620 v2 at 2.10GHz with 78GB RAM, GeForce GTX 1080 (with 8GB GDDR) GPU and 800GB Intel NVMe SSD DC P3700 with 2.8GB/s sequential read throughput. We use Ubuntu 16.04.3 with kernel 4.4.15 that includes GAIA modifications, CUDA SDK 8.0.34, and NVIDIA-UVM driver 384.59.

Performance cost of functional emulation. The emulation introduces performance penalties that do not exist in the CPU analogues of the emulated functions. In particular, `unmapV` constitutes more than 99% of `macquire` latency, and `fetchPageModifiedBit` occupies nearly 100% of `mrelease`.

These functions are expensive only because of the lack of the appropriate GPU driver and hardware support. We expect them to be as fast as their CPU analogues if implemented by GPU vendors. For example, `mmap` or `mprotect` calls for a 1GB region take less than 10 μseconds on the CPU. If implemented for the GPU, they might last slightly longer due to over-PCIe access to update the page tables.

To be safe, we conservatively assume that `unmapV` and `fetchPageModifiedBit` are as slow as *10 msec* in all the reported results. These are the worst-case estimates, yet they allow us to provide a *reliable estimate* of GAIA performance in future systems.

Evaluation methodology. We run each experiment 11 times, omit the first result as a warmup, and report the average. We flush the system page cache before each run (unless stated otherwise). We do not report standard deviations below 5%.

Overhead analysis

Impact on CPU I/O. GAIA introduces additional version checks into the CPU I/O path. To measure the overheads, we run the standard TIO benchmark suite [23] for evaluating CPU I/O performance. We run random/sequential reads/writes using the default configuration with 256KB I/O requests, accessing a 1GB file. As a baseline, we run the benchmark on the unmodified Linux kernel 4.4.15.

We vary the number of supported GPUs in the system as it affects the number of version checks. We observe less than 1% and up to 5% overhead for 32GPUs and 160GPUs respectively for random reads, and no measurable overheads for sequential accesses. We conclude that *GAIA introduces negligible performance overheads for legacy CPU file I/O*.

Memory overheads. The main dynamic cost stems from pristine page copies for 3-way merge. However, common read-only workloads require no such copies, therefore incurring no extra memory cost. Otherwise, the memory overheads depend on the write intensity of the workload. GAIA creates one copy for every writable GPU page in the system.

The static cost is due to the addition of version vectors to the page cache. This cost scales linearly with the utilized

[3] GAIA source code is publicly available at https://github.com/acsl-technion/GAIA.

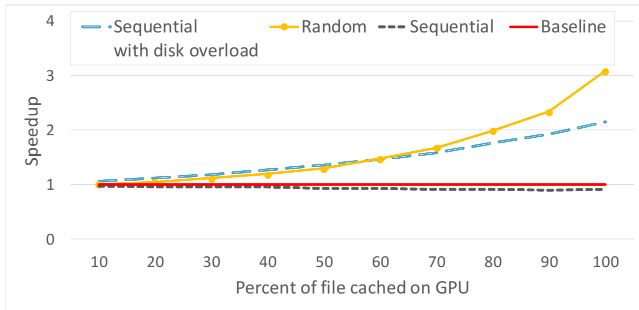


Figure 5: CPU I/O speedup of GAIA over unmodified Linux. Higher is better.

GPU memory size as GPU versions are stored only for GPU-resident pages, but grows quadratically with the number of GPUs. With 512 bytes per version vector entry, the worst-case memory overhead (all GPUs fill their memories with files) and 100 GPUs each with 8GB memory, the overhead reaches about 5GB, which is less than 1% of the total utilized GPU memory (800GB).

Microbenchmarks

Benefits of peer caching. We measure the performance of *CPU* POSIX `read` accesses to a 1GB file which is partially cached in GPU memory. Thus, these accesses result in reading the GPU-resident pages from the GPU instead of the SSD. We vary the cached portion of the file, reading the rest from the SSD. We run three experiments: (1) random reads (2) sequential reads, and (3) sequential reads while the SSD is busy serving other I/O requests. We compare to unmodified Linux, where all the data is fetched from the SSD.

Figure 5 shows that peer-caching can boost the CPU performance by up to 3× for random reads, and up to 2× when the SSD is under load. GAIA is no faster than SSD for sequential reads, however. This is due to the GPU DMA controller bottleneck, stemming from the lack of public interfaces to program its scatter-gather lists. Therefore, GAIA is forced to perform the I/O one GPU page at a time.

False sharing. We run the same CPU-GPU false-sharing microbenchmark as in §3.1. We map the shared buffer from a file, and let the CPU and the GPU update it concurrently with non-overlapping writes. GAIA merges the page when the GPU kernel terminates. We compare with the execution where each of the processors writes into a private buffer, without false sharing. We observe that GAIA with shared buffer is *the same* as the baseline. The cost of merging the page in the end is constant per 4KB page: 1.4 μsecond.

This experiment highlights the fact that *GAIA eliminates the overheads of false sharing entirely*.

Streaming read performance. We evaluate the case where the GPU reads and processes the whole file. We use three kernels: (1) a zero-compute kernel that copies the data from the input pointer into an internal buffer, one stride at a time per threadblock; (2) an unmodified LUD kernel, representative of compute-intensive kernels in Rodinia benchmark suite [19]; (3) a *closed-source* cuBLAS SGEMM library kernel [7]. We modify their CPU code such that the GPU input is read from a file rather than from memory, as in prior work [40].

We evaluate several techniques to read files into the GPU:

1. **CUDA-[must fit in GPU memory]**: read from the host, copy to GPU;
2. **GPUfs-[requires GPU code changes]**: read from the GPU kernel via GPUfs API. GPUfs uses 64KB pages as in GPU hardware;
3. **UVM**: read from the host into UVM memory (physically residing in CPU), read from the GPU kernel;
4. **GAIA**: map the file into GPU, read from the GPU kernel.

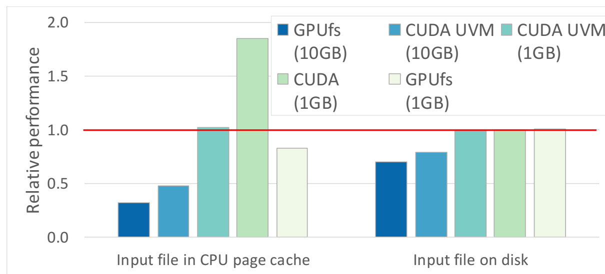
We implement all four variants for the zero-compute kernel. LUD and cuBLAS cannot run with GPUfs because that requires changing their code. We run the experiments with two input files, one smaller and one larger than GPU memory.

Figure 6a shows the results of reading a 1GB and 10GB file for the zero-compute kernel. GAIA is competitive with UVM and GPUfs for all of the evaluated use cases, but slower than CUDA when working with a small file in the CPU page cache, due to inefficiency of the GPU data transfers in GAIA. In CUDA, the data is copied in a single transfer over PCIe, whereas in GAIA the I/O is broken into multiple non-consecutive 64KB blocks, which is much slower.

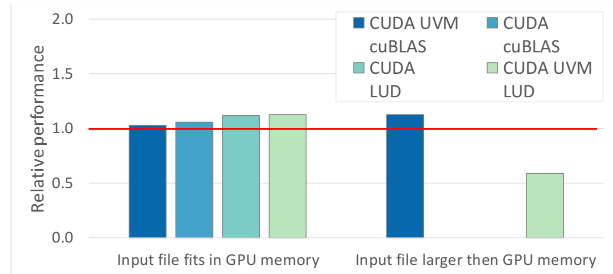
GAIA is faster than UVM when reading cached files due to the UVM's extra data copy, and faster than GPUfs with 1GB because of GPUfs trashing. The thrashing occurs because the GPUfs buffer cache in GPU memory is not large enough to store the whole file. Pages are constantly evicted as a result, introducing high runtime overheads.

Figure 6b shows the results of processing a 1GB and a 10GB file for the compute-intensive LUD and cuBLAS kernels for techniques (1), (3) and (4). Large files cannot be evaluated with CUDA as they do not fit in GPU memory. GAIA is faster than LUD on UVM, yet slightly slower than the other methods. With cuBLAS, the specific access pattern results in a large working set, causing trashing with GAIA.

The insufficient coordination with the OS I/O prefetcher in the current implementation makes it slower when the file fits GPU memory. The performance can be improved via a more sophisticated prefetcher design and transfer batching in future GPU drivers. With large files, however, *GAIA is on par with and faster than UVM, while offering the convenience of using a GPU-agnostic OS API and supporting unmodified GPU kernels.*



(a) Zero-compute kernel, normalized to GAIA. Higher is better.



(b) LUD and cuBLAS small and large files (from disk). Higher is better.

Figure 6: Streaming read I/O performance analysis

	Prefetched	On disk
GAIA (sec)	1.2	2.9
UVM (sec)	11.4 (↑ 9×)	17.8 (↑ 6×)
ActivePointers(4 CPU threads) (sec)	0.5 (↓ 2×)	1.7 (↓ 2×)
ActivePointers(1 CPU thread) (sec)	0.6 (↓ 2×)	5.5 (↑ 2×)

Table 2: Image collage: GAIA vs. UVM vs. ActivePointers

Applications

Performance of on-demand data I/O

We use an open-source image collage benchmark [34]. It processes an input image by replacing its blocks with "similar" tiny images from a large indexed dataset stored in a file. The access pattern depends on the input: each block of the input image is processed separately to generate the index into the dataset, fetching the respective tiny image afterward. The dataset is 19GB, thus it does not fit in memory of our GPU. While only about 25% of the dataset is required to completely process one input, the accesses are input-dependent.

We compare three implementations: (1) original ActivePointers, (2) UVM and (3) GAIA. For the last two we modify the original code by replacing ActivePointers [34] with regular pointers. In UVM the dataset is first read into a shared UVM buffer. In GAIA the file is `mmap`-ed into GPU memory.

Both ActivePointers and GAIA allow random file access from the GPU, but they differ in that they rely on software-emulated and hardware page faults respectively.

Table 2 shows the end-to-end performance comparison. GAIA is 9× and 6× faster than UVM, because it accesses the data in the file on-demand, whereas in UVM the file must be read in full prior to kernel invocation.

We investigate the performance difference between ActivePointers and GAIA. We observe that ActivePointers use four I/O threads on the CPU to serve GPU I/O requests. Reducing the number of I/O threads to only one as in GAIA provides a more fair comparison. In this case, GAIA is 2× faster when reading data from disk, but still 2× slower when the file is prefetched. The reasons are not yet clear, however.

We conclude that GAIA's on-demand data access is competitive with highly optimized ActivePointers, and *significantly faster* than UVM.

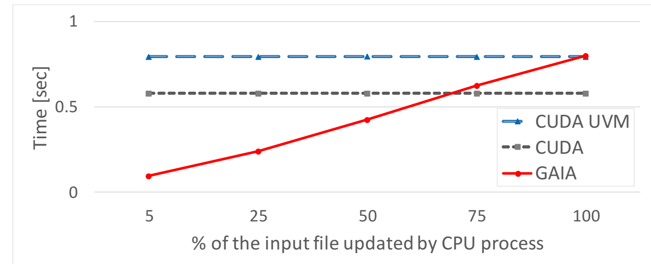


Figure 7: Graph processing with dynamic graph updates, while varying the number of updates. Lower is better.

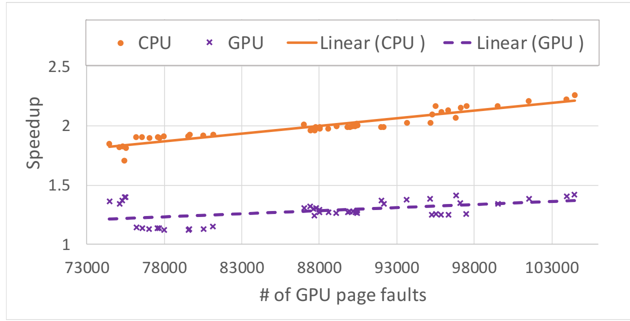
Dynamic graph processing with Gunrock

We focus on a scenario where graph computations are invoked multiple times, but the input graph periodically changes. This is the case, for example, for a road navigation service such as Google Maps, which needs to accommodate the traffic changes or road conditions while responding to user queries.

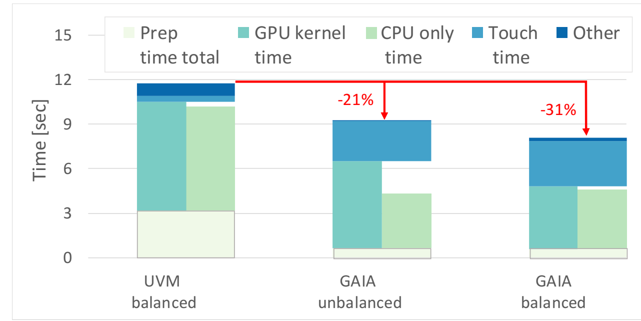
We assume the following service design. There are two processes: an updater daemon (producer) and a GPU-accelerated compute server (consumer), which share the graph database. The daemon running on the CPU (1) retrieves the graph updates (traffic status) from an external server; (2) updates the graph database file; and (3) signals to the compute service to recalculate the routes. The latter reads these updates from the file each time it recomputes in response to a user query. The producer updates only part of the graph (i.e., edge weights representing traffic conditions). This design is modular, easy to implement, and supports very large datasets. Similar producer-consumer scenarios have also been used in prior work [15].

We use an *unmodified* Gunrock library [38] for fast graph processing. We run the Single Source Shortest Path algorithm provided with Gunrock, modifying it to read input from a file, which is updated in a separate (legacy) CPU process that uses standard I/O (no consistency-synchronizing system calls). The file updates and graph computations are interleaved: the compute server invokes the updater, which in turn invokes the computations, and so on. We run a loop of 100 iterations and measure the total running time.

We implement the benchmark using UVM and GAIA, and also compare it with the original CUDA implementation. For



(a) CPU and GPU speedups over UVM while varying the number of computations. Higher is better.



(b) End-to-end runtime comparison. Other: memory (de)/allocation. Touch time = time of first access from CPU, includes merge for GAIA. Prep time = time of reading input. Lower is better.

Figure 8: Performance impact of false sharing in image stitching.

both UVM and CUDA, the *whole file* must be copied into a GPU-accessible buffer on every update because the locations of the modified data in the file are unknown. No such copy is required for GAIA, where each GPU kernel invocation is surrounded by `macquire` and `mrelease` calls.

We run the experiment on the uk_2002 input graph provided with Gunrock examples, extended to include edge weights. The file size is 5.2GB.

Figure 7 shows the performance as a function of the portion of the graph being updated. GAIA is faster than the alternatives with fewer changes to the file, automatically detecting the changes in the pages that were indeed modified. For the worst case of full file update (above 75%) GAIA becomes slower than the original CUDA implementation. This is due to the inefficiency of the GPU data transfers, as we also observed in Fig. 6a. This experiments shows *the utility of GAIA's fine-grain consistency control, which enables efficient computations in a read-write sharing case.*

Effects of false sharing in image stitching

We consider an image processing application used in optical microscopy to acquire large images. Microscopes acquire these images as grids of overlapping patches that are then stitched together. Recent works accelerate this process on multiple GPUs [18, 16]. The output image is split into non-overlapping sub-images, where each GPU produces its output independently, and the final result is merged into a single output image.

This application benefits from using GAIA to write into the shared output file directly from GPUs, eliminating the need to store large intermediate buffers in CPU memory. We seek to show the effects of false sharing in this workload if it were implemented with GAIA. Unfortunately, we cannot fully evaluate GAIA on multiple GPUs, as we explained earlier (§5.2.3). Instead, we implement only its I/O-intensive component in the CPU and the GPU.

Both the CPU and GPU load their patches, with already pre-computed output coordinates. Each patch is sharpened via a convolution filter, and then is written to the output file. The convolution filter is invoked several times per output to explore a range of different compute loads. In GAIA, we map both the input patches and the output file into the CPU and the GPU. For the UVM baseline, the inputs are read into UVM memory before kernel invocation.

We run the experiment on a public Day2 Plate [3] stitching dataset with 5.3GB of input tiles and 1.3GB of output. We use the patch locations included in the dataset to write the patches, which ensures realistic access to the output file.

We split the output image over the vertical dimension and load-balance the input such that both the CPU and the GPU run about the same time in the baseline implementation which writes into a UVM shared output buffer. The patch coordinates determine the amount of false sharing in this application.

Figure 8a shows the speedup of GAIA over UVM while varying the amount of computations per patch. We observe up to 45% speedup for the CPU, and over 2.3× speedup for the GPU. This experiment corroborates the conclusions of the microbenchmark in (§3.1), now in a realistic write-sharing workload. GAIA enables *significant performance gain by eliminating false sharing in the unified page cache*.

To evaluate the complete system rather than the performance of each processor separately, we pick one of the runtime parameters in the middle of Figure 8a, and measure the end-to-end runtime, including file read, memory allocation, and page merging. We prefetch the input into the page cache on the CPU to highlight the performance impact.

Figure 8b shows the results. For exactly the same runtime configuration GAIA *outperforms UVM by 21%*. Moreover, GAIA allows further improvements by rebalancing the load between the processors (GPU runs faster without false sharing), achieving overall 31% performance improvement.

Related work

To the best of our knowledge, GAIA is the first system to offer a distributed page cache abstraction for GPUs that is integrated into the OS. Our work builds on prior research in

the following areas.

Memory coherence for distributed systems. Lazy Release Consistency [22, 10] serves as the basis for GAIA. GAIA implements the home-based version of LRC [41]. Munin [14] implements eager RC by batching the updates. GAIA adopts this idea by using LRC-dirty bit to batch multiple updates. Version Vectors is an idea presented in [32] for detecting mutual inconsistency in multiple-writers systems. We believe that GAIA is the first to apply these ideas to building a heterogeneous OS page cache.

Heterogeneous, multi-core, and distributed OS design for systems without cache coherence. Several proposed OSes support heterogeneous and non-cache coherent systems by applying distributed system principles to their design [29, 13, 12]. None of these systems implements shared page cache support, which is the main tenet of our work.

K2 [25] is a shared-most OS for mobile devices running over several coherence domains. It implements a sequentially consistent software DSM for the OS structures shared among the domains. K2 DSM implements sequential consistency, which is a strict coherence model. Similarly to GAIA, K2 relies on page faults as the trigger for consistency operations. K2 DSM implementation of the coherence model relies heavily on the underlying hardware. Thus, even read-only sharing is not possible as it requires a different MMU for handling reads and writes. GAIA does not have this limitation. Solros [27] proposes a data-centric operating system to enable efficient I/O access for XeonPhi accelerators. Solros includes a buffer cache for faster I/O, but unlike GAIA, it is limited to host memory, and does not explicitly discuss inter-accelerator sharing.

The file system in the FOS multikernel [39] shares data between cores but is limited to read-only workloads. Hare [21] is a file system for non-cache-coherent multicores in which each node may cache file data in the private memory and a shared global page cache. Hare uses close-to-open semantics, which GAIA refines. Distributed OSes such as Sprite [30], Amoeba [28], and MOSIX [11] aim to provide a single system image abstraction and in particular, coherent and transparent access to files from different nodes, but they achieve this via process migration/use home nodes to forward their I/O.

GPU file I/O. GPUfs [36] allows file access from GPU programs and implements a distributed weakly consistent page cache with session semantics. ActivePointers [34] extend GPUfs with a software-managed address translation and page faults, enabling GPU memory mapped files. However, unlike GAIA, ActivePointers require intrusive changes to GPU kernels, its session semantics is too coarse-grain for our needs, and its page cache is not integrated with the CPU page cache, thus lacks peer-caching support. SPIN [15] integrates direct GPU-SSD communications into the OS. As in GAIA peer-caching, SPIN adds a mechanism for choosing the best path for file I/O to the GPU. However, it does not extend the page cache into the GPU.

Memory management in GPUs. NVIDIA Unified Virtual Memory (UVM) and Heterogeneous Memory Management (HMM) [4] allow both the CPU and GPU to share virtual memory, migrating the pages to/from GPU/CPU upon page fault. Neither currently supports memory mapped files on x86 processors. Both introduce a strict coherence model that suffers from false sharing overheads. Asymmetric Distributed Shared Memory [20] is a precursor of UVM that emulates a unified address space between CPUs and GPUs in software.

IBM Power9 CPU with NVIDIA V100 GPUs provides hardware support for coherent memory between the CPU and the GPU. Since GPU memory is managed as another NUMA node, memory-mapped files are naturally accessible from the GPUs. This approach would not work for commodity x86 architectures which lack CPU-GPU hardware coherence.

Dragon [26] extends NVIDIA UVM to enable GPU access to large data sets residing in NVM storage, by mapping them into the GPU address space. Dragon focuses exclusively on accessing the NVM from the GPU, does not integrate GPU memory into a unified page cache, has no peer-caching support, and does not consider CPU-GPU file sharing, all of which are the main contributions of our work.

Conclusions

GAIA enables GPUs to map files into their address space via a weakly consistent page cache abstraction over GPU memories that is fully integrated with the OS page cache. This design optimizes both CPU and GPU I/O performance while being backward compatible with legacy CPU and unmodified GPU kernels. GAIA's implementation in Linux for NVIDIA GPUs shows promising results for a range of realistic application scenarios, including image and graph processing. It demonstrates the benefits of lazy release consistency for write-shared workloads.

GAIA demonstrates the importance of integrating GPU memory in the OS page cache, and proposes the minimum set of memory management extensions required for future OSes to provide the unified page cache services introduced by GAIA.

Acknowledgments

We thank Idit Keidar, Isaac Gelado and our shepherd David Nellans for their valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18).

References

[1] 'Beyond GPU Memory Limits with Unified Memory on Pascal'. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/.

[2] CUDA toolkit documentation - cudaStreamAddCallback(). https://docs.nvidia.com/cuda/

cuda-runtime-api/group__CUDART__STREAM.html.

[3] Data dissemination: Reference Image Stitching Data. https://isg.nist.gov/deepzoomweb/data/referenceimagestitchingdata.

[4] Heterogeneous Memory Management (HMM). https://www.kernel.org/doc/html/v4.18/vm/hmm.html.

[5] IEEE 1003.1-2001 - IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(R)). https://standards.ieee.org/standard/1003_1-2001.html.

[6] Radeon's next-generation Vega architecture. https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf.

[7] cuBLAS Library User Guide. https://docs.nvidia.com/pdf/CUBLAS_Library.pdf, October 2018.

[8] Everything you need to know about Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-/know-about-unified-memory.pdf, February 2018.

[9] NVIDIA Tesla V100 GPU accelerator data sheet. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf, March 2018.

[10] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[11] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[12] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[13] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44. ACM, 2009.

[14] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. *SIGPLAN Notices*, 25(3):168–176, February 1990.

[15] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 167–179, Santa Clara, CA, 2017. USENIX Association.

[16] Timothy Blattner, Walid Keyrouz, Joe Chalfoun, Bertrand Stivalet, Mary Brady, and Shujia Zhou. A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images. In *2014 43rd International Conference on Parallel Processing*, pages 1–9, Sept 2014.

[17] William J. Bolosky and Michael L. Scott. False Sharing and Its Effect on Shared Memory Performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, volume 57, 1993.

[18] Joe Chalfoun, Michael Majurski, Tim Blattner, Kiran Bhadriraju, Walid Keyrouz, Peter Bajcsy, and Mary Brady. MIST: Accurate and Scalable Microscopy Image Stitching Tool with Stage Modeling and Error Minimization. *Scientific reports*, 7(1):4988, 2017.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[20] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 347–358. ACM, 2010.

[21] Charles Gruenwald III, Filippo Sironi, M Frans Kaashoek, and Nickolai Zeldovich. Hare: A File System for Non-cache-coherent Multicores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 30:1–30:16. ACM, 2015.

[22] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21. ACM, 1992.

[23] Mika Kuoppala. Tiobench-threaded I/O bench for Linux, 2002.

[24] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, November 1989.

[25] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. *ACM SIGARCH Computer Architecture News*, 42(1):285–300, 2014.

[26] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. DRAGON: breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 32. IEEE Press, 2018.

[27] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proceedings of the Thirteenth EuroSys Conference*, page 36. ACM, 2018.

[28] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[29] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 221–234. ACM, 2009.

[30] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23–36, February 1988.

[31] Tom Papatheodore. Summit System Overview. https://www.olcf.ornl.gov/wp-content/uploads/2018/05/Intro_Summit_System_Overview.pdf, June 2018.

[32] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[33] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Feb 2007.

[34] Sagi Shahar, Shai Bergman, and Mark Silberstein. ActivePointers: A Case for Software Address Translation on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 596–608, June 2016.

[35] Sagi Shahar and Mark Silberstein. Supporting Data-driven I/O on GPUs Using GPUfs. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 12:1–12:11. ACM, 2016.

[36] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498. ACM, 2013.

[37] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[38] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A High-performance Graph Processing Library on the GPU. *SIGPLAN Notices*, 51(8):11:1–11:12, February 2016.

[39] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, April 2009.

[40] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 13–24. IEEE Computer Society, 2015.

[41] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 75–88. ACM, 1996.

Transkernel: Bridging Monolithic Kernels to Peripheral Cores

Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin

Purdue ECE

Abstract

Smart devices see a large number of ephemeral tasks driven by background activities. In order to execute such a task, the OS kernel wakes up the platform beforehand and puts it back to sleep afterwards. In doing so, the kernel operates various IO devices and orchestrates their power state transitions. Such kernel executions are inefficient as they mismatch typical CPU hardware. They are better off running on a low-power, microcontroller-like core, i.e., peripheral core, relieving CPU from the inefficiency.

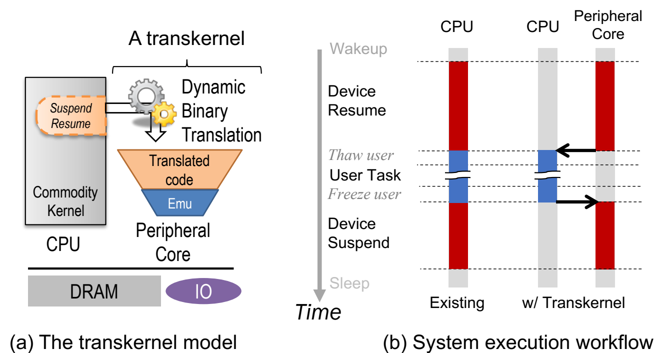
We therefore present a new OS structure, in which a lightweight virtual executor called *transkernel* offloads specific phases from a monolithic kernel. The transkernel translates stateful kernel execution through cross-ISA, dynamic binary translation (DBT); it emulates a small set of stateless kernel services behind a narrow, stable binary interface; it specializes for hot paths; it exploits ISA similarities for lowering DBT cost.

Through an ARM-based prototype, we demonstrate transkernel's feasibility and benefit. We show that while cross-ISA DBT is typically used under the assumption of efficiency *loss*, it can enable efficiency *gain*, even on off-the-shelf hardware.

1 Introduction

Driven by periodic or background activities, modern embedded platforms[1] often run a large number of ephemeral tasks. Example tasks include acquiring sensor readings, refreshing smart watch display [44], push notifications [38], and periodic data sync [91]. They drain a substantial fraction of battery, e.g., 30% for smartphones [13, 12] and smart watches [45], and almost the entire battery of smart things for surveillance [84]. To execute an ephemeral task, a commodity OS kernel, typically implemented in a monolithic fashion,

[1] This paper focuses on battery-powered computers such as smart wearables and smart things. They run commodity OSes such as Linux and Windows. We refer to them as embedded platforms for brevity.



(a) The transkernel model (b) System execution workflow

Figure 1: An overview of this work

drives the whole hardware platform out of deep sleep beforehand (i.e., "resume") and puts it back to deep sleep afterwards (i.e., "suspend"). During this process, the kernel consumes much more energy than the user code [44], up to 10× shown in recent work [38].

Why is the kernel so inefficient? Recent studies [26, 92, 44] show the bottlenecks as two kernel phases called *device suspend/resume* as illustrated in Figure 1. In the phases, the kernel operates a variety of IO devices (or *devices* for brevity). It invokes device drivers, cleans up pending IO tasks, and ensures devices to reach expected power states. The phases encompass concurrent execution of drivers, deferred functions, and hardware interrupts; they entail numerous CPU idle epochs; their optimization is proven difficult (§2) [92, 50, 46].

We deem that device suspend/resume mismatches CPU. It instead would be much more efficient on low-power, microcontroller-like cores, as exemplified by ARM Cortex-M. These cores are already incorporated as *peripheral cores* on a wide range of modern system-on-chips (SoCs) used in production such as Apple Watch [59] and Microsoft Azure Sphere [52]. On IO-intensive workloads, a peripheral core delivers is much more efficient than the CPU due to lower idle power and higher execution efficiency [42, 43, 54, 1]. Note that running *user code* (which often builds atop POSIX) on peripheral cores is a non-goal: on one hand, doing so would gain much less efficiency due to fewer idle epochs in user

execution; on the other hand, doing so requires to support a much more complex POSIX environment on peripheral cores.

Offloading the execution of a commodity, monolithic kernel raises practical challenges, not only i) that the peripheral core has a different ISA and wimpy hardware but also ii) that the kernel is complex and rapidly evolving [68]. Many OS proposals address the former while being inadequate in addressing the latter [5, 61, 6, 48, 75]. For instance, one may refactor a monolithic kernel to span it over CPU and a peripheral core; the resultant kernel, however, depends on a wide binary interface (ABI) for synchronizing state between the two ISAs. This interface is brittle. As the upstream kernel evolves, maintaining *binary compatibility* across different ISAs inside the kernel itself soon becomes unsustainable. Instead, we argue for the code running on peripheral cores to enjoy firmware-level compatibility: developed and compiled *once*, it should work with *many* builds of the monolithic kernel – generated from different configurations and source versions.

Our response is a radical design called *transkernel*, a lightweight virtual executor empowering a peripheral core to run specific kernel phases – device suspend/resume. Figure 1 overviews the system architecture. A transkernel executes unmodified kernel binary through cross-ISA, dynamic binary translation (DBT), a technique previously regarded as expensive [5] and never tested on microcontroller-like cores to our knowledge. Underneath the translated code, a small set of emulated services act as lightweight, drop-in replacements for their counterparts in the monolithic kernel. Four principles make transkernel practical: i) translating stateful code while emulating stateless kernel services; ii) identifying a narrow, stable translation/emulation interface; iii) specializing for hot paths; iv) exploiting ISA similarities for DBT.

We demonstrate a transkernel prototype called ARK (**A**n a**R**m trans**K**ernel). Atop an ARM SoC, ARK runs on a Cortex-M3 peripheral core (with only 200 MHz clock and 32KB cache) alongside Linux running on a Cortex-A9 CPU. ARK transparently translates unmodified Linux kernel drivers and libraries. It depends on a binary interface consisting of only 12 Linux kernel functions and one kernel variable, which are stable for years. ARK offers complete support for device suspend/resume in Linux, capable of executing diverse drivers that implement rich functionalities (e.g., DMA and firmware loading) and invoke sophisticated kernel services (e.g., scheduling and IRQ handling). As compared to native kernel execution, ARK only incurs 2.7× overhead, 5.2× lower than a baseline of off-the-shelf DBT. ARK reduces system energy by 34%, resulting in tangible battery life extension under real-world usage.

We make the following contributions on OS and DBT:

- We present the transkernel model. In the design space of OSes for heterogeneous multi-processors, the transkernel represents a novel point: it combines DBT and emulation for bridging ISA gaps and for catering to core asymmetry, respectively.
- We present a transkernel implementation, ARK. Targeting Linux, ARK presents specific tradeoffs between kernel translation versus emulation; it identifies a narrow interface between the two; it contributes concrete realization for them.
- Crucial to the practicality of ARK, we present an inverse paradigm of cross-ISA DBT, in which a microcontroller-like core translates binary built for a full-fledged CPU. We contribute optimizations that systematically exploit ISA similarities. Our result demonstrates that while cross-ISA DBT is typically used under the assumption of efficiency *loss*, it can enable efficiency *gain*, even on off-the-shelf hardware.

The source code of ARK can be found at http://xsel.rocks/p/transkernel.

2 Motivations

We next discuss device suspend/resume, the major kernel bottleneck in ephemeral tasks, and that it can be mitigated by running on a peripheral core. We show difficulties in known approaches and accordingly motivate our design objectives.

2.1 Kernel in device suspend/resume

Expecting a long period of system inactivity, an OS kernel puts the whole platform into deep sleep: in brief, the kernel synchronizes file systems with storage, freezes all user tasks, turns off IO devices (i.e., device suspend), and finally powers off the CPU. To wake up from deep sleep, the kernel performs a mirrored procedure [11]. In a typical ephemeral task, the above kernel execution takes hundreds of milliseconds [31] while the user execution often takes tens of milliseconds [44]; the kernel execution often consumes several times more energy than the user execution [38].

Problem: device suspend/resume By profiling recent Linux on multiple embedded platforms, our pilot study [92] shows the aforementioned kernel execution is bottlenecked by device suspend/resume, in which the kernel cleans up pending IO tasks and manipulates device power states. The findings are as follows. i) ***Device suspend/resume is inefficient.*** It contributes 54% on average and up to 66% to the total kernel energy consumption. CPU idles frequently in numerous short epochs, typically in milliseconds. ii) ***Devices are diverse.*** On a platform, the kernel often suspends and resumes tens of different devices. Across platforms, the bottleneck devices are different. iii) ***Optimization is difficult.*** Device power state transitions are bound by slow hardware and low-speed buses, as well as physical factors (e.g., voltage ramp-up). While Linux already parallelizes power transitions with great efforts [50, 46], many power transitions must happen sequentially per *implicit* dependencies of power, voltage, and clock. As a result, CPU idle constitutes up to 68% of the device suspend/resume duration.

SoC	Cores	ISAs	Shared DRAM?	Mapping kern mem?	Shared IRQ
OMAP4460 [83] (2010)	A9+M3	v7a+v7m	Full	Yes. MPU	39/102
AM572x [81] (2014)	A15+M4	v7a+v7m	Full	Yes. MPU	32/92
i.MX6SX [62] (2015)	A9+M4	v7a+v7m	Full	Yes. MPU	85/87
i.MX7 [65] (2017)	A7+M4	v7a+v7m	Full	Yes. MPU	88/90
i.MX8M [63] (2018)	A53+M4	v8a+v7m	Full	Yes. MPU	88/88
MT3620 [52] (2018)*	A7+M4	v7a+v7m	Full	Likely. MPU	Likely most

Table 1: Our hardware model fits many popular SoCs which are used in popular products such as Apple Watch and Azure Sphere. Section 7.5 discusses caveats. *: lack public technical details.

Challenge: Widespread, complex kernel code Device suspend/resume invokes multiple kernel layers [68, 32]. Specifically, it invokes functions in individual drivers (e.g., MMC controllers), driver libraries (e.g., the generic clock framework), kernel libraries (e.g., for radix trees), and kernel services (e.g., scheduler). In a recent Linux source tree (4.4), we find that over 1000 device drivers, which represent almost all driver classes, implement suspend/resume callbacks in 154K SLoC. These callbacks in turn invoke over 43K SLoC in driver libraries, 8K SLoC in kernel libraries, and 43K SLoC in kernel services. The execution is control-heavy, with dense branches and callbacks.

Opportunities We observe the following kernel behaviors in device suspend/resume. i) ***Low sensitivity to execution delay*** On embedded platforms, most ephemeral tasks are driven by background activities [38, 53, 13]. This contrasts to many servers for interactive user requests [93, 53]. ii) ***Hot kernel paths*** In successful suspend/resume, the kernel acquires all needed resources and encounters no failures [41]. Off the hot paths, the kernel handles rare events such as races between IO events, resource shortage, and hardware failures. These branches typically cancel the current suspend/resume attempt, perform diagnostics, and retry later. Unlike hot paths, they invoke very different kernel services, e.g., syslog. iii) ***Simple concurrency*** exists among the syscall path (which initiates suspend/resume), interrupt handlers, and deferred kernel work. The concurrency is for hardware asynchrony and kernel modularity rather than exploiting multicore parallelism.

Summary: design implications Device suspend/resume shall be treated systematically. We face challenges that the invoked kernel code is diverse, complex, and cross-layer; we see opportunities that allow focusing on hot kernel paths, specializing for simple concurrency, and gaining efficiency at the cost of increased execution time.

2.2 A peripheral core in a heterogeneous SoC

Hardware model We set to exploit peripheral cores already on modern SoCs. Hence, our software design only assumes the following hardware model which fits a number of popular SoCs as listed in Table 1.

1. ***Asymmetric processors***: In different coherence domains, the CPU and the peripheral core offer disparate performance/-efficiency tradeoffs. The peripheral core has memory protection unit (MPU) but no MMU, incapable of running commodity OSes as-is.

2. ***Heterogeneous, yet similar ISAs***: The two processors have different ISAs, in which many instructions have similar *semantics*, as will be discussed below.

3. ***Loose coupling***: The two processors are located in separate power domains and can be turned on/off independently.

4. ***Shared platform resources***: Both processors share access to platform DRAM and IO devices. Specifically, the peripheral core, through its MPU, should map all the kernel code/-data at identical virtual addresses as the CPU does. Both processors must be able to receive interrupts from the devices of interest, e.g., MMC; they may, however, see different interrupt line numbers of the same device.

How can peripheral cores save energy? They are known to deliver high efficiency for IO-heavy workloads [42, 54, 78, 1, 76]. Specifically, they benefit the kernel's device suspend/resume in the following ways. i) A peripheral core can operate while leaving the CPU offline. ii) The idle power of a peripheral core is often one order of magnitude lower [43, 64], minimizing system power during core idle periods. iii) Its simple microarchitecture suits kernel execution, whose irregular behaviors often see marginal benefits from powerful microarchitectures [58]. Note that a peripheral core offers much higher efficiency than a LITTLE core as in ARM big.LITTLE [24], which mandates a homogeneous ISA and tight core coupling. We will examine big.LITTLE in Section 7.

ISA similarity On an SoC we target, the CPU and the peripheral core have ISAs from the same family, e.g., ARM. The two ISAs often implement similar instruction *semantics* despite in different *encoding*. The common examples are SoCs integrating ARMv7a ISA and ARMv7m ISA [62, 65, 81, 52, 83]. Other families also provide ISAs amenable to same-SoC integration, e.g., NanoMIPS and MIPS32. We deem that the ISA similarities are *by choice*. i) For ISA designers, it is feasible to explore performance-efficiency tradeoffs within one ISA family, since the family choice is merely about instruction *syntax* rather than *semantics* [8]. ii) For SoC vendors, incorporating same-family ISAs on one chip simplifies software efforts [40], silicon design, and ISA licensing.

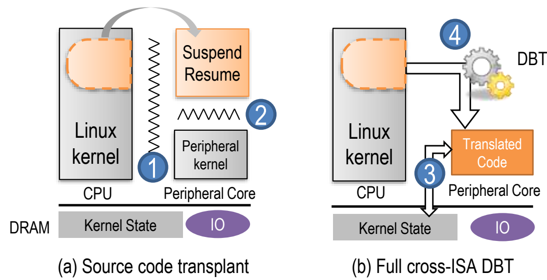


Figure 2: Alternative ways for offloading kernel phases

Count	Category
359	Device specific
845	Driver lib
217	Kernel lib
858	Kernel services

(a) # of functions

From \ To	v2.6	v3.16	v4.4	v4.9	v4.17
v2.6 (Jan 2011)		155 378	196 385	194 384	213 395
v3.16 (Aug 2014)	500 717		155 674	163 661	214 707
v4.4 (Jan 2016)	640 855	216 780		55 721	159 828
v4.9 (Dec 2016)	816 938	354 848	214 797		173 848
v4.17 (Jul 2018)	1075 1111	606 1043	498 1060	384 1015	

(b) # of functions (upper) & types (lower) w/ changed ABI across kernel versions

Figure 3: Counts of Linux kernel functions referenced by device suspend/resume, showing (a) the functions are rich and diverse and (b) their ABI change is substantial over time. Exported functions only. Build config: omap2defconfig. ABI changes detected with ABI compliance checker [70]

2.3 OS design space exploration

We set to realize heterogeneous execution for an *existing* monolithic kernel.

How about refactoring the kernel and cross-compiling statically? One may be tempted to modify a monolithic kernel (we use Linux as the example below) [43, 5] to be one unified source tree; the tree shall be cross-compiled into a kernel binary for CPU and a "peripheral kernel" for the peripheral core. This approach results in an OS structure shown in Figure 2(a). Its key drawback is the two interfaces that are difficult to implement and maintain, shown as ∿∿∿ in the figure.

① The interface between two heterogeneous ISAs, as needed for resolving inter-kernel data dependency. Through the interface, both kernels synchronize their kernel state, e.g., devices configurations, pending IO tasks, and locks, before and after the offloading. Built atop shared memory [43, 5, 23], the interface is essentially an agreement on thousands of shared Linux kernel data types, including their semantics and/or memory layout. The agreement is brittle, as it is affected by ISA choices, kernel configurations, and kernel versions. Hence, keeping data types consistent across ISAs entails tedious tweak of kernel source and configurations [22, 23]. As Greg Kroah-Hartman puts, "you will go insane over time if you try to support this kind of release, I learned this the hard way a long time ago." [36]

② The interface between the transplant code and the peripheral kernel, as needed for resolving functional dependency. In principle, this interface is determined by the choice of transplant boundary. In prior work, the example choices include the interface of device-specific code [22, 23, 80], that of driver classes [10, 79], or that of driver libraries [43]. All these choices expose at least hundreds of Linux kernel functions on this interface, as summarized in Figure 3(a). This is due to Linux's diverse, sophisticated drivers. Implementing such an interface is daunting; maintaining it is even more difficult due to significant ABI changes [37] as shown in Figure 3(b).

In summary, all these difficulties root in the peripheral kernel's *deep dependency* on the Linux kernel. This is opposite to the common practice: heterogeneous cores to run their own "firmware" that has little dependency on the Linux kernel. This is sustainable because the firmware stays compatible with many builds of Linux.

How about virtual execution? Can we minimize the dependency? One radical idea would be for a peripheral core to run the Linux kernel through virtual execution, as shown in Figure 2(b). Powered by DBT, virtual execution allows a *host* processor (e.g., the peripheral core) to execute instructions in a foreign *guest* ISA (e.g., the CPU). Virtual execution is free of the above interface difficulties: the translated code precisely reproduces the kernel behaviors and directly operates the kernel state (③). The peripheral core interacts with Linux through a low-level, stable interface: the CPU's ISA (④).

The problem, however, is the high overhead of existing cross-ISA DBT [4]. It is further exacerbated by our *inverse* DBT paradigm: whereas existing cross-ISA DBT is engineered for a brawny host emulating a weaker guest (e.g., an x86 desktop emulating an ARM smartphone) [17, 87], our DBT host, a peripheral core, shall serve a full-fledged CPU. A port of popular DBT exhibits up to 25× slowdown as will be shown in §7. Such overhead would negate any efficiency promised by the hardware and result in overall efficiency *loss*. Furthermore, cross-ISA DBT for the *whole* Linux kernel is complex [7]. A peripheral core lacks necessary environment, e.g., multiple address spaces and POSIX, for developing and debugging such complex software.

2.4 Design objective

We therefore target threefold objective.

G1. Tractable engineering. We set to reuse much of the kernel source, in particular the drivers that are impractical to build anew. We target simple software for peripheral cores.

G2. Build once, work with many. One build of the peripheral core's software should work with a commodity kernel's binaries built from a wide range of configurations and source

versions. This requires the former to interact with the latter through a stable, narrow ABI.

G3. Low overhead. The offloaded kernel phases should yield a tangible efficiency gain.

3 The Transkernel Model

Running on a peripheral core, a transkernel consists of two components: a DBT engine for translating and executing the unmodified kernel binary; a set of emulated, minimalistic kernel services that underpin the translated kernel code, as will be described in detail in Section 4. A concrete transkernel implementation targets a specific commodity kernel, e.g., Linux. A transkernel does not execute user code in ephemeral tasks as stated in Section 1.

The transkernel follows four principles:

1. Translating stateful code; emulating stateless services By *stateful code*, we refer to the offloaded code that must share states with the kernel execution on CPU. The stateful code includes device drivers, driver libraries, and a small set of kernel services. They cover the most diverse and widespread code in device suspend/resume (§2). By translating their binaries, the transkernel reuses the commodity kernel without maintaining wide, brittle ABIs. (objective G1, G2)

The transkernel emulates a tiny set of kernel services. We relax their semantics to be stateless, so that their states only live within one device suspend/resume phase. Being stateless, the emulated services do not need to synchronize states with the kernel on CPU over ABIs. (G2)

2. Identifying a narrow, stable translation/emulation ABI The ABI must be unaffected by kernel configurations and unchanged since long in the kernel evolution history. (G2)

3. Specializing for hot paths In the spirit of OS specialization [20, 71, 51], the transkernel only executes the hot path of device suspend/resume; in the rare events of executing off the hot path, it transparently falls back on CPU. The transkernel's emulated services seek *functional equivalence* and only implement features needed by the hot path; they do not precisely reproduce the kernel's behaviors. (G1)

4. Exploiting ISA similarities for DBT The transkernel departs from generic cross-DBT that bridges arbitrary guest/host pairs; it instead systematically exploits similarities in instructions semantics, register usage, and control flow transfer. This makes cross-ISA DBT affordable. (G3)

Limitations First, across ISAs of which instruction semantics are substantially different, e.g., ARM and x86, the transkernel may see diminishing or even no benefit. Second, the transkernel's longer delays (albeit lower energy) may misfit latency-sensitive contexts, e.g., for waking up platforms in response to user input. Our current prototype relies on heuristics to recognize such contexts and falls back on the CPU accordingly (Section 4).

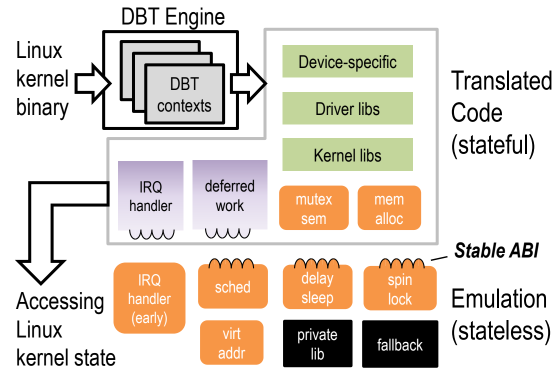


Figure 4: The ARK structure on a peripheral core

In Section 4 below we describe how to apply the model to a concrete transkernel, in particular our translation/emulation decisions for major kernel services, and our choices of the emulation interface. We will describe DBT in Section 5.

4 ARK: An ARM Transkernel

Targeting an ARM SoC, we implement a transkernel called ARK. The SoC encompasses a popular combination of ISAs: ARMv7A for its CPU and ARMv7m for its peripheral core. The CPU runs Linux v4.4.

Offloading workflow ARK is shipped as a standalone binary for the peripheral core, accompanied by a small Linux kernel module for control transfer between CPU and the peripheral core. We refer to such control transfer as *handoff*. Prior to a device suspend phase, the kernel shuts down all but one CPU cores, passes control to the peripheral core, and shuts down the last CPU core. Then, ARK completes the device phase in order to suspend the entire platform. Device resume is normally executed by ARK on the peripheral core; in case of urgent wakeup events (e.g., a user unlocking a smart watch screen), the kernel resumes on CPU with native execution.

System structure As shown in Figure 4, ARK runs a DBT engine, its emulated kernel services, and a small library for managing the peripheral core's private hardware, e.g., interrupt controllers. The emulated services serves downcalls (⌒⌒⌒) from the translated code and makes upcalls (~⌣⌣⌣) into the translated code. Table 2 summarizes the interfaces. Upon booting, ARK replicates Linux kernel's linear memory mappings for addressing kernel objects in shared memory [43, 23]. ARK maps I/O regions with MPU and time-multiplexes the regions on the MPU entries.

To support concurrency in the offloaded kernel phases, ARK runs multiple DBT contexts. Each context has its own DBT state (e.g., virtual CPU registers and a stack), executing DBT and emulated services independently. Context switch is as cheap as updating the pointer to the DBT state.

ARK executes the hot paths. Upon entering cold branches pre-defined by us, e.g., kernel `WARN()`, ARK migrates all the

Kernel services	Implementations & reasons
Scheduler (§4.1)	Emulated. Reason: simple concurrency.
IRQ handler (§4.2)	Early stage emulated; then translated
HW IRQ controller (§4.2)	Emulated. Reason: core-specific
Deferred work (§4.3)	Translated. Reason: stateful
Spinlocks (§4.4)	Emulated. Reason: core-specific
Sleepable locks (§4.4)	Fast path translated. Reason: stateful
Slab/Buddy allocator (§4.5)	Fast path translated. Reason: stateful
Delay/wait/jiffies (§4.6)	Emulated. Reason: core-specific

jiffies	udelay()	msleep()	tasklet_schedule()	irq_thread()
ktime_get()	queue_work_on()	worker_thread()	run_local_timers()	
generic_handle_irq()	schedule()	async_schedule()*	do_softirq()*	

*=ABI unchanged since 2014 (v3.16); others unchanged since 2011 (v2.6).

Table 2: Top: Kernel services supported by ARK. Bottom: Linux kernel ABI (12 funcs+1 var) ARK depends on. ARK offers complete support for device suspend/resume in Linux.

DBT contexts of *translated* code back to the CPU and continues as *native* execution there (§6).

4.1 A Scheduler of DBT Contexts

ARK emulates a scheduler which shares no state, e.g., scheduling priorities or statistics, with the Linux scheduler on the CPU. Corresponding to the simple concurrency model of suspend/resume (§2), ARK eschews reproducing Linux's preemptive multithreading but instead maintains and switches among cooperative DBT contexts: one primary context for executing the syscall path of suspend/resume, one for executing IRQ handlers (§4.2), and multiple for deferred work (§4.3). Managing no more than tens of contexts, ARK uses simple, round-robin scheduling. It begins the execution in the syscall context; when the syscall context blocks (e.g., by calling `msleep()`), ARK switches to the next ready context to execute deferred functions until they finish or block. When an interrupt occurs, ARK switches to the IRQ context to execute the kernel interrupt handler (§4.2).

4.2 Interrupt and Exception Handling

During the offloaded device phase, all interrupts are routed to the peripheral core and handled by ARK.

Kernel interrupt handlers ARK emulates a short, early stage of interrupt handling while translating the kernel code for the remainder. This is because this early stage is ISA-specific (e.g., for manipulating the interrupt stack), on which the CPU (v7a) and the peripheral core (v7m) differ. Hence, the emulated services implement a v7m-specific routine and install it as the hardware interrupt handler. Once an interrupt happens, the routine is invoked to finish the v7m-specific task and make an upcall to the kernel's ISA-neutral interrupt handling routine (listed in Table 2), from where the ARK translates the kernel to finish handling the interrupt.

Hardware interrupt controller ARK emulates the CPU's hardware interrupt controller. This is needed as the two cores have separate, heterogeneous interrupt controllers. The CPU controller's registers are unmapped in the peripheral core; upon accessing them (e.g., for masking interrupt sources) the translated code triggers faults. ARK handles the faults and operates the peripheral core's controller accordingly.

Exception: unsupported We don't expect any exception in the offloaded kernel phases. In case exception happens, ARK uses its fallback mechanism (§6) to migrate back to CPU.

4.3 Deferred Work

Device drivers frequently schedule functions to be executed in the future. ARK translates the Linux services that schedule the deferred work as well as the actual execution of the deferred work. ARK chooses to translate such services because they must be *stateful*: the peripheral core may need to execute deferred work created on the CPU prior to the offloading, e.g., freeing pending WiFi packets; it may defer new work until after the completion of resume.

ARK maintains dedicated DBT contexts for executing the deferred work (Section 4.1). While the Linux kernel often executes deferred work in kernel threads (daemons), our insight is that deferred work is oblivious to its execution context (e.g., a real Linux thread or a DBT context in ARK). Beyond this, ARK only has to run the deferred work that may *sleep* with separate DBT contexts so that they do not block other deferred work. From these DBT contexts, ARK translates the main functions of the aforementioned kernel daemons, which retrieve and invoke the deferred work.

Threaded IRQ defers heavy-lifting IRQ work (i.e., deferred work) to a kernel thread which executes the work after the hardware IRQ is handled. A threaded IRQ handler may sleep. Therefore, ARK maintains per-IRQ DBT contexts for executing these handlers. Each context makes upcalls into `irq_thread()` (the main function of threaded irq daemon, listed in Table 2).

Tasklets, workitems, and timer callbacks The kernel code may dynamically submit short, non-sleepable functions (tasklets) or long, sleepable functions (workitems) for deferred execution. Kernel daemons (softirq and kworker) execute tasklets and workitems, respectively.

ARK creates one dedicated context for executing all non-sleepable tasklets and per-workqeueue contexts for executing workitems so that one workqueue will not block others. These contexts make upcalls to the main functions of the kernel daemons (`do_softirq()`, `worker_thread()`, and `run_local_timers()`), translating them for retrieving and executing deferred work.

4.4 Locking

Spinlocks ARK emulates spinlocks, because their implementation is core-specific and that ARK can safely assume all spinlocks are free at handoff points: as described in early

Section 4, handoff happens between one CPU core and one peripheral core, which do not hold any spinlock; all other CPU cores are offline and cannot hold spinlocks. Hence, ARK emulates spinlock acquire/release by pausing/resuming interrupt handling. This is because ARK runs on one peripheral core and the only hardware concurrency comes from interrupts.

Sleepable locks ARK translates sleepable locks (e.g., mutex, semaphore) because these locks are stateful: for example, the kernel's clock framework may hold a mutex preventing suspend/resume from concurrently changing clock configuration [56]. Furthermore, mutex's seemingly simple interface (i.e., compare & exchange in fast path) has *unstable* ABI and therefore unsuitable for emulation: a mutex's reference count type changes from `int` to `long` (v4.10), breaking the ABI compatibility. The translated operations on sleepable locks may invoke spinlocks or the scheduler, e.g., when updating reference counts or putting the caller to sleep, for which the translated execution makes downcalls to the emulated services.In practice, no sleepable lock is held prior to system suspend.

4.5 Memory Allocation

The device phase frequently requests dynamic memory, often at granularities of tens to hundreds of bytes. By Linux design, such requests are served by the kernel slab allocator backed by a buddy system for page allocation (fast path); when the physical pages runs low, the kernel may trigger swapping or kill user processes (slow path).

ARK provides memory allocation as a stateful service. It translates the kernel code for the fast path, including the slab allocator and the buddy system. In the case that the allocation enters the slow path (e.g., due to low physical memory), ARK aborts offloading; fortunately, our stress test suggests such cases to be extremely rare, as will be reported in Section 7. With a stateful allocator, the offloaded execution can free dynamic memory allocated during the kernel execution on CPU, and vice versa. Compare to prior work that instantiates per-kernel allocators with split physical memory [43], ARK reduces memory fragmentation and avoids tracking *which* processor should free *what* dynamic memory pieces. Our experience in Section 7 show that ARK is able to handle intensive memory allocation/free requests such as in loading firmware to a WiFi NIC.

4.6 Delays & Timekeeping

Delays ARK emulates `udelay()` and `msleep()` for busy waiting and sleeping. ARK converts the expected wait time to the hardware timer cycles on the peripheral core. ARK implements `msleep()` by pausing scheduling the caller context.

jiffies The Linux kernel periodically updates jiffies, a global integer, as a low-overhead measure of elapsed time. By consulting the peripheral core's hardware timer, ARK directly updates the jiffies. It is thus the only shared variable on the kernel ABI that ARK depends (all others are functions).

5 The Cross-ISA DBT Engine

A Cross-ISA DBT Primer DBT, among its other uses [60, 49, 27], is a known technique allowing a *host* processor to execute instructions in a foreign *guest* ISA. In such cross-ISA DBT, the host processor runs a program called DBT engine. At run time, the engine reads in guest instructions, translates them to host instructions based on the engine's built-in *translation rules*, and executes these host instructions. The engine translates guest instructions in the unit of translation block – a sequence (typically tens) of guest instructions that has one entry and one or more exits. After translating a block, the engine saves the resultant host instructions to its *code cache* in the host memory, so that future execution of this translated block can be directed to the code cache.

Design overview We build ARK atop QEMU [7], a popular, opensource cross-ISA DBT engine. ARK inherits QEMU's infrastructure but departs from its generic design which translates between arbitrary ISAs. ARK targets two well-known DBT optimizations: i) to emit as few host instructions as possible; ii) to exit from the code cache to the DBT engine as rarely as possible. We exploit the following similarities between the CPU's and the peripheral core's ISAs (ARMv7a & ARMv7m):

1. Most v7a instructions have v7m counterparts with identical or similar semantics, albeit in different encoding. (§5.1)
2. Both ISAs have the same general purpose registers. The condition flags in both ISAs have same semantics. (§5.2)
3. Both ISAs use program counter (PC), link register (LR), and stack pointer (SP) in the same way. (§5.3)

Beyond the similarities, the two ISAs have important discrepancies. Below, we describe our exploitation of the ISA similarities and our treatment for *caveats*.

5.1 Exploiting Similar Instruction Semantics

	Category	Cnt	v7m
w/ CNTPRT	Identity	447	1
	Side effect	52	3-5
	Const constraints	22	2-5
	Shift modes	10	2
w/o counterparts		27	2-5
Total (v7a)		558	

Table 3: Translation rules for v7a instructions. Column 3: the number of v7m instructions emitted for one v7a instruction

We devise translation rules with a principled approach by parsing a machine-readable, formal ISA specification recently published by ARM [72]. Our overall guideline is to map each v7a instruction to one v7m instruction that has identical or similar semantics. We call them *counterpart* instructions. For a counterpart instruction with similar (yet

ARMv7a	ARMv7m (by ARK)
G1: `ldr r0, [r1], r2, lsr #4`	**H1:** `ldr.w r0, [r1]` **H2:** `lsr.w t0, r2, 0x4` **H3:** `add.w r1, r1, t0`
G2: `adds r0, r1, 0x80000001`	**H4:** `mov.w t0, 0xc0` **H5:** `ror.w t0, t0, 0x7` **H6:** `adds.w r0, r1, t0`
G3: `sub r0, r1, r2`	**H7:** `sub.w r0, r1, r2`

Table 4: Sample translation by ARK. By contrast, our baseline QEMU port translates G1–G3 to **27** v7m instructions

non-identical) semantics, ARK emits a few "amendment" v7m instructions to make up for the semantic gap. The resultant translation rules are based on individual guest instructions, different from translation rules based on one or more translation blocks commonly seen in cross-ISA DBT [86]. This is because semantics similarities allows identity translation for most guest instructions. Amendment instructions are oblivious to interrupts/exceptions: as stated in §4.2, ARK defers IRQ handling to translation block boundary and expects no exceptions.

Table 3 summarizes ARK's translation rules for all 558 v7a instructions. Among them, 80% can be translated with identity rules, for which ARK only needs to convert instruction encoding at run time. 15% of v7a instructions have v7m counterparts but may require amendment instructions, which fortunately fall into a few categories: i) ***Side effects.*** After load/store, v7a instructions may additionally update memory content or register values (shown in Table 4, G1). ARK emits amendment instructions to emulate the extra side effect (H3). ii) ***Constraints on constants.*** The range of constants that can be encoded in a v7m instruction is often narrower (Table 4, G2). In such cases, the amendment instructions load the constant to a scratch register, operate it, and emulate any side effects (e.g., index update) the guest instruction may have. iii) ***Richer shift modes.*** v7a instructions support richer shift modes and larger shift ranges than their v7m counterparts. This is exemplified by Table 4 G1, where a v7m instruction cannot perform LSR (logic shift right) inline as its v7a counterpart. Similar to above, the amendment instructions perform shift on the operand in a scratch register.

Beyond the above, only 27 v7a instructions have no v7m counterparts, for which we manually devise translation rules.

In summary, through systematic exploitation of similar instruction semantics, ARK emits compact host code at run time. In the example shown in Table 4, three v7a instructions are translated into seven v7m instructions by ARK, while to 27 instructions by our QEMU baseline.

5.2 Passthrough of CPU registers

General purpose registers Both the guest (v7a) and the host (v7m) have the same set (13) of general-purpose registers. In allocating registers of a host instruction, ARK follows guest register allocation with best efforts (e.g., one-to-one mapping in best case, as in Table 4, G1). ARK emits much fewer host instructions than QEMU, which emulates all guest registers in host memory with load /store.

Caveats fixed The amendment host instructions operate scratch registers as exemplified by `t0` in Table 4, H2-H6. However, the wimpy host faces higher register pressure, as it (v7m) has no more registers than the brawny guest (v7a). To spill some registers to memory while still reusing the guest's register allocation, we make the following tradeoff: we designate one host register as the *dedicated* scratch register, and emulates its guest counterpart register in memory. We pick the *least* used one in the guest binary as the dedicated scratch register, which is experimentally determined as R10 by analyzing kernel binary. We find most amendment instructions are satisfied by *one* scratch register; in rare cases when extra scratch registers are needed, ARK follows a common design to allocate dead registers and spill unused ones to memory.

Condition flags Both the guest and the host ISAs involve five hardware condition flags (e.g., zero and carry) with identical semantics; fortunately, most guest (v7a) instructions and their host (v7m) counterparts have identical behaviors in testing/setting flags per the ISA specifications [72]. ARK hence directly emits instructions to manipulate the host's corresponding flags. Such flag passthrough especially benefits control-heavy suspend/resume, which contains extensive conditional branches (§2); we study its benefits quantitatively in §7.3.

Caveats fixed Amendment host instructions may affect the hardware condition flags unexpectedly. For amendment instructions (notably comparison and testing) that *must* update the flags as mandated by ISA, ARK emits two host instructions to save/restore the flags in a scratch register around the execution of these amendment instructions.

5.3 Control Transfer and Stack Manipulation

Function call/return Both guest (v7a) and host (v7m) use PC (program counter) and LR (link register) to maintain the control flow. QEMU emulates guest PC and LR in host memory. As a result, the return address, loaded from stack or the emulated LR, points to a guest address (i.e., kernel address). Each function return hence causes the DBT to step in and look up the corresponding code cache address. This overhead is magnified in the control-heavy device phase.

By contrast, ARK never emits host code to emulate the guest (i.e., kernel) PC or LR. For each kernel function call, ARK saves the return addresses within *code cache* on stack or in LR; for each kernel function return, ARK loads the return address (which points to code cache) to hardware PC from the stack or the hardware LR. By doing so, ARK no longer participates in all function returns. Our optimization is inspired by same-ISA DBT [34].

Stack and SP QEMU emulates the guest (i.e., kernel) stack and SP with a host array and a variable. Each guest push/pop translates to multiple host instructions updating the stack array and the emulated SP. This is costly, as suspend/resume frequently makes function calls and operates stack heavily.

ARK avoids such expensive stack emulation by emitting host push/pop instructions to directly operate the guest stack *in place*. This is possible because ARK emulates the Linux kernel's virtual address space (§4). ARK also ensures the host code generate the same stack frames as the guest would do by making amendment instructions avoid using stack, which would introduce extra stack contents. In addition, this further facilitates the migration in abort (§6).

Caveats fixed i) As the host saves on the guest stack the code cache addresses, which are meaningless to the guest CPU, upon migrating from the peripheral core (host) to the CPU (guest), the DBT rewrites all code cache addresses on stack with their corresponding guest addresses. ii) guest push/pop instruction may involve emulated registers (i.e., scratch register). ARK must emit multiple host instructions to correctly synchronize the emulated registers in memory.

6 Translated ⟶ Native Fallback

As described in Section 3, when going off the hot paths, ARK migrates the kernel phase back to the CPU and continues as native execution, analogous to virtual-to-physical migration of VMs [85]. Migrating one DBT context is natural, as ARK passes through most CPU registers and uses the kernel stack in place (§5.3). Yet, to migrate *all* active DBT contexts, ARK address the following unique challenges.

Migrate DBT contexts for deferred work After fallback, all blocked workitems should continue their execution on the CPU. Unfortunately, their enclosing DBT contexts do not have counterparts in the Linux kernel. To solve this issue, we again exploit the insight that the workitems are oblivious to their execution contexts. Upon migration, the Linux kernel creates temporary kernel threads as "receivers" for blocked workitems to execute in. Once the migrated workitems complete, the receiver threads terminate.

Migrate DBT context for interrupt If fallback happens inside an ISA-neutral interrupt handler (translated), the remainder of the handler should migrate to the CPU. This challenge, again, is that ARK's interrupt context has no counterpart on the CPU: the interrupt never occurs to the CPU. ARK addresses this by *rethrowing* the interrupt as an IPI (interprocessor interrupt) from the peripheral core to the CPU; the Linux kernel uses the IPI context as the receiver for the migrated interrupt handler to finish execution.

Section 7 will evaluate the fallback frequency and cost.

7 Evaluation

We seek to answer the following questions:

1. Does ARK incur tractable engineering efforts? (§7.2)
2. Is ARK correct and low-overhead? (§7.3)
3. Does ARK yield energy efficiency benefit? What are the major factors impacting the benefit? (§7.4)

7.1 Methodology

Test Platform We evaluate ARK on OMAP4460, an ARM-based SoC [83] as summarized in Table 6. We chose this SoC mainly for its good documentation and longtime kernel support (since 2.6.11), which allows our study of kernel ABI over a long timespan in Section 2. As Cortex-M3 on the platform is incapable of DVFS, for fair comparison, we run both cores at their highest clock rates. Note that OMAP4460 is not completely aligned with our hardware model, for which we apply workarounds as will be discussed in Section 7.5.

Benchmark setup We benchmark ARK on the whole suspend/resume kernel phases. We run a user program as the test harness that periodically kicks ARK for suspend/resume; the generated kernel workloads are the same as in all ephemeral tasks. Our benchmark is macro: it exercise extensive drivers and services, during which ARK translates and executes over 200 million instructions.

The benchmark operates nine devices for suspend/resume. 1. **SD card**: SanDisk Ultra 16GB SDHC1 Class 10 card; 2. **Flash drive**: a generic drive connected via USB; 3. **MMC controller**: on-chip OMAP HSMMC host controller; 4. **USB controller**: on-chip OMAP HS multiport USB host controller; 5. **Regulator**: TWL6030 power management IC connected via I2C; 6. **Keyboard**: Dell KB212-B keyboard connected via USB; 7. **Camera**: Logitech c270 connected via USB; 8. **Bluetooth NIC**: an adapter with Broadcom BCM20702 chipset connected via USB; 9. **WiFi NIC**: TI WL1251 module. The kernel invokes sophisticated drivers, thoroughly exercising various services including deferred work (2–4,6–8), slab/buddy allocator (1–4,6–9), softirq (9), DMA (2,6–9), threaded IRQ (1,5,9), and firmware upload (9).

We measure device suspend/resume executed by ARK on Cortex-M3 and report the measured results. We compare ARK to native Linux execution on Cortex-A9. We further compare to a baseline ARK version: its DBT is a straightforward v7m port of QEMU that misses optimizations described in Section 5. We report measurements taken with warm DBT code cache, as this reflects the real-world scenario where device suspend/resume is frequently exercised.

7.2 Analysis of engineering efforts

ARK eliminates source refactoring of the Linux kernel (§2.3). As shown in Table 5, ARK transparently reuses substantial kernel code (15K SLoC in our test), most of which are drivers

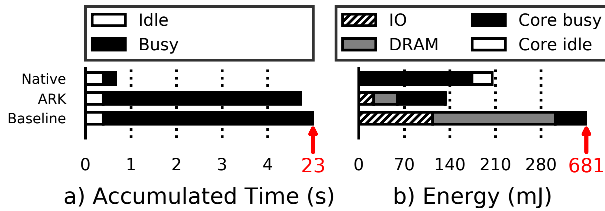


Figure 5: Execution time and energy in device suspend/resume. ARK substantially reduces the energy.

and their libraries. We stress that ARK, as a driver-agnostic effort, not only enables reuse of the drivers under test but also other drivers in the ARMv7 Linux kernel.

Table 5 also shows that ARK requires modest efforts in developing new software for the peripheral core. The 9K new SLoC for DBT is low as compared to commodity DBT (e.g., QEMU has 2M SLoC). ARK implements emulation in as low as 1K SLoC and in return avoids translating generic, sophisticated Linux kernel services [34, 21]. The result validates our principle of specializing these emulated services.

Existing code (unchanged)	
Translated	15K SLoC
Substituted w/ emu	25K SLoC
New implementation	
DBT	9K SLoC
Emulation	1K SLoC

Table 5: Source code

ARK meets our goal of "build once, run with many". We verify that the ARK binary works with a variety of kernel configuration variants (including `defconfig-omap4` and `yes-to-all`) of Linux 4.4. We also verify that ARK works with a wide range of Linux versions, from version 3.16 (2014) to 4.20 (most recent at the time of writing). This is because ARK only depends on a narrow ABI shown in Table 2, which has not changed since Linux 3.16.

7.3 Measured execution characteristics

ARK's correctness Formally, we derive translation rules from the specification of ARM ISA [72]; experimentally, we validate ARK by comparing its execution results side-by-side with native execution and examining the translated code with the native kernel binary. Over 200 million executed instructions, we verify that ARK's translation preserves kernel's semantics and presents consistent execution results.

Core activity We trace core states during ARK execution. Figure 5 (a) shows the breakdown of execution time. Compared to the native execution on CPU, ARK shows the same amount of accumulated idle time but much longer (16×) busy time. The reasons are Cortex-M3's much lower clock rate (1/6 of the A9's clock rate) and ARK's execution overhead. Despite the extended busy time, ARK still yields energy benefit, as we will show below.

Memory activity We collect DRAM activities by sampling

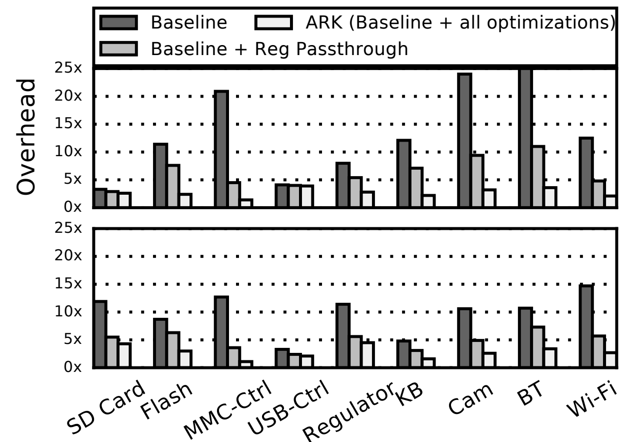


Figure 6: Busy execution overhead for devices under test (top: suspend; bottom: resume). Our DBT optimizations reduce the overhead by up to one order of magnitude

the hardware counters of the SoC's DDR controller. We observed that ARK on Cortex-M3 generates much higher average DRAM utilization (32 MB/s read and 2MB/s write) than the native execution on A9 (only 8MB/s read and 4MB/s write). We attribute such thrashing to M3's small (32KB) last-level cache (LLC). Throughout the test, the ARK emitted and executed around 230KB host instructions, which far exceeds the LLC capacity and likely causes thrashing. By contrast, Cortex-A9 has a much larger LLC (1MB), which absorbs most of the kernel memory access. The memory activity has a strong energy impact, as will be shown below.

Busy execution overhead Our measurement shows that ARK incurs low overhead in busy kernel execution, which includes both DBT and emulation. We report the overhead as the ratio between ARK's cycle count on Cortex-M3 to the Linux's cycle count on A9. Note that an M3 cycle is 6× longer than A9 due to different clock rates.

Overall, the execution overhead is 2.7× on average (suspend: 2.9×; resume: 2.6×). Of individual drivers, the execution overhead ranges from 1.1× to 4.5× as shown in Figure 6. Our DBT optimizations (§5) have strong impact on lowering the overhead. Lacking them, our baseline design incurs a 13.9× overhead on average, 5.2× higher than ARK. We examined how our optimizations contribute to the gap: register passthrough (§5.2) reduces the baseline's overhead by 2.5× to 5.5×. Remaining optimizations (e.g., control transfer) collectively reduce the overhead by additional 2×. Our optimizations are less effective on drivers with very dense control transfer (e.g., USB) due to high DBT cost.

Emulated services Our profiling shows that ARK's emulated services incur low overhead. Overall, the emulated services only contribute 1% of total busy execution. i) The early, core-specific interrupt handling (§4.2) takes 3.9K Cortex-M3 cycles, only 1.5–2× more cycles than the native execution on

	CPU	Peripheral core
Core	Cortex A9@1.2GHz	Cortex M3@200MHz
Cache	L1:64KB + L2:1MB	L1:32KB
Typical busy/idle power	630mW/80mW	17mW/1mW

Table 6: The test platform - OMAP4460 on a Pandaboard

A9. ii) Emulated workqueues (§4.3) incurs a typical queueing delay of tens of thousands M3 cycles. The delay is longer than the native execution but does not break the deferred execution semantics.

Fallback frequency & cost We stress test ARK by repeating the benchmark for 1000 runs. Throughout the 1000 runs, ARK encounters only four cases when the execution goes off the hot path, all of which caused by the WiFi hardware failing to respond to resume commands; it is likely due to an existing glitch in WiFi firmware. In such a case, ARK migrates execution by spending around 20 us on rewriting code cache addresses on stack (§5.3), 17 us to flush Cortex-M3's cache, and 2 us to wake up the CPU through an IPI.

7.4 Energy benefits

Methodology We study system-level energy and in particular how it is affected by ARK's its extended execution time. We include energy of both cores, DRAM, and IO.

We measure power of cores by sampling the inductors on the power rails for the CPU and the peripheral core. As the board lacks measurement points for DRAM power [19], we model DRAM power as a function of DRAM power state and read/write activities, with Micron's official power model for LPDDR2 [55]. The system energy of ARK is given by:

$$E_{ARK} = \underbrace{E_{core}}_{\text{Measured}} + T_{idle} \cdot \underbrace{(P_{mem_sr} + P_{io})}_{\textit{Modeled}} + T_{busy} \cdot \underbrace{(P_{mem} + P_{io})}_{\textit{Modeled}}$$

Here, E_{core} is the measured core energy. All Ts are measured execution time. Ps are power consumptions for DRAM and IO: P_{mem} is DRAM's active power derived from measured DRAM activities as described in Section 7.3; P_{mem_sr} is DRAM's self-refresh power, 1.3mW according to the Micron model; P_{io} is the average IO power which we estimate as 5mW based on prior work [90]. Note that during suspend/resume, IO devices no longer actively perform work, thus consuming much less power.

Energy saving ARK consumes 66% energy (a reduction of 34%) of the native execution, despite its longer execution time. The energy breakdown in Figure 5(b) shows the benefit comes from two portions: i) in busy execution, ARK's energy efficiency is 23% higher than the native execution due to low overhead (on average 2.7×); ii) during system idle, ARK reduces system energy to a negligible portion, as the peripheral core's idle power is only 1.25% of the CPU's. Figure 5(b) highlights the significance of our DBT optimizations: the baseline, like ARK, benefits from lower idle power as well; however its high execution overhead ultimately leads to 5.1× energy compared to the native execution. Interestingly, ARK

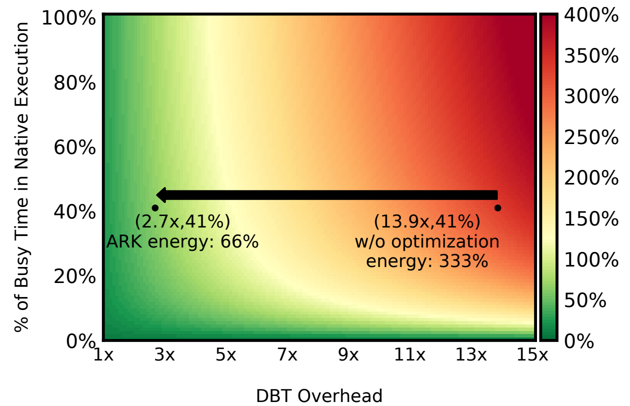


Figure 7: System energy consumption (inc. cores, DRAM, and IO) of ARK relative to native execution (100%), under different DBT overheads (x-axis) and processor core usage (y-axis). ARK's low energy hinges on low DBT overhead.

consumes *more* DRAM energy than the native execution. We deem the cause as Cortex-M3's tiny LLC (32KB) as describe earlier. Our result suggests that the current size is suboptimal for the offloaded kernel execution.

What-if analysis How sensitive is ARK's energy saving to two major factors: the DBT overhead (ARK's behavior) and the processor core usage (Linux's behavior)? To answer the question, we estimate the *what-if* energy consumption by using the power model as described above. The analysis results in Figure 7 show two findings. i) ARK's energy benefit will be more pronounced with lower core usage (i.e., longer core idle), because ARK's efficiency advantage over native execution is higher during core idle. ii) ARK's energy benefit critically depends on DBT. When the DBT overhead (on x-axis) drops to below 3.5×, ARK saves energy even for 100% busy execution; when the overhead exceeds 5.2×, ARK wastes energy even for 20% busy execution, the lowest core usage observed on embedded platforms in prior work [92].

Qualitative comparison with big.LITTLE We estimate ARK saves tangible energy compared to a LITTLE core. We use parameters based on recent big.LITTLE characterizations [66, 25]: compared to the big (i.e., CPU on our platform), a LITTLE core has 40 mW idle power [69] and offers 1.3× energy efficiency at 70% clock rate [47]. We favorably assume LITTLE's DRAM utilization is as low as the big, while in reality the utilization should be higher due to LITTLE's smaller LLC. Even with this favorable assumption for LITTLE and unfavorable, tiny LLC for ARK, LITTLE consumes 77% energy of native execution, more than ARK (51%–66%), mainly because LITTLE's idle power is 40× of Cortex-M3. Furthermore, ARK's advantage will be even more pronounced with a proper LLC as discussed earlier.

Battery life extension Based on ARK's energy reduction in device suspend/resume, we project the battery life extension for ephemeral tasks reported in prior work [38]. When the ephemeral tasks are executed at 5-second intervals and the

native device suspend/resume consumes 90% system energy in a wakeup cycle, ARK extends the battery life by 18% (4.3 hours per day); with 30-second task intervals and a 50% energy consumption percentage, ARK extends the battery life by 7% (1.6 hours per day). This extension is tangible compared to complementary approaches in prior work [38, 90].

7.5 Discussions

Workarounds for OMAP4460 While OMAP4460 mostly matches our hardware model as summarized in Table 1, for minor mismatch we apply the following workarounds. ***Memory mapping*** Our hardware model (§2.2) mandates that the peripheral core should address the entire kernel memory. Yet, Cortex-M3, according to ARM's hardware specification [82], is only able to address memory in certain range (up to `0xE0000000`), which unfortunately does not encompass the Linux kernel's default address range. As a workaround, we configure the Linux kernel source, shifting its address range to be addressable by Cortex-M3. ***Interrupt handling*** While our hardware model mandates that both processors should receive all interrupts, OMAP4460 only routes a subset of them (39/102) to Cortex-M3, leaving out IO devices such as certain GPIO pins. These IO devices hence are unsupported by the ARK prototype and are not tested in our evaluation.

Recommendation to SoC architects To make SoCs friendly to a transkernel, architects may consider: i) routing all interrupts to CPU and the peripheral core, ideally with the identical interrupt line numbers; ii) making the peripheral core capable of addressing the whole memory address space; iii) enlarging the peripheral core's LLC size modestly. We expect a careful increase (e.g., to 64 KB or 128 KB) will significantly reduce DRAM power at a moderate overhead in the core power.

Applicability to a pair of 64-bit/32-bit ISAs While today's smart devices often use ARMv7 CPUs, emerging ARM SoCs start to combine 64-bit CPUs (ARMv8) with 32-bit peripheral core (ARMv7m), as listed in Table 1. On one hand, transkernel's idea of exploiting ISA similarity still applies, as exemplified by G2→H2 in Table 7; on the other hand, its DBT overhead may increase significantly for the following reasons. Compared to the 32-bit ISA, the 64-bit ISA has richer instruction semantics, more general purpose registers, and a much larger address space. As a result, ARK cannot pass through 64-bit CPU registers but instead have to emulate them in memory; ARK must translate the guest's 64-bit memory addresses to 32-bit addresses supported by the host (Table 7 G1→H1), e.g., by keeping consistent two sets of page tables, for 64-bit and 32-bit virtual address spaces, respectively; with large physical memory (>4GB), even this technique will not work because the peripheral core's page tables are incapable of mapping the extra physical memory.

ARMv8	ARMv7m (by ARK, ideally)

G1:
 	
`ldrb w2, [x22, #1059]`
 	
`ldrb w1, [x0, #160]`
	H1:
	`(emulate x22+#1059 in addr1)`
	`ldrb r2, [addr1]`
	`(emulate x0+#160 in addr2)`
	`ldrb r1, [addr2]`

G2: `cmp w2, w1`	H2: `cmp r2, r1`
G3: `beq mmc_select_bus_width+0x160`	H3: `beq mmc_select_bus_width+0x160`

Table 7: Ideal AARCH64 translation by ARK for `mmc_compare_ext_csds()` in Linux v4.4. While identity mapping still exists (G2→H2), software emulation can diminish ARK's benefits (G1→H1).

8 Related Work

OS for heterogeneous cores A multikernel OS runs its kernels on individual processors. A number of such OSes are designed anew with a strong distributed system flavor. They define explicit message interfaces among kernels [6, 88, 2]; some additionally exploit managed languages/compilers to generate such interfaces [61]. Unlike them, transkernel targets spanning an *existing* monolithic kernel and therefore adopts DBT to address the resultant interface challenge.

OSes like Popcorn [5] and K2 [43] seek to present a single Linux image over heterogeneous processors. For sharing kernel state across ISAs, they rely on manual source tweaks or hand-crafted communication. They face the interface difficulty as described in §2.3.

Prior systems distribute OS functions over CPU and accelerators [57, 77]. The accelerators cannot operate autonomously, which is however required by device suspend/resume. Prior systems offload apps from a smartphone (weak) to cloud servers (strong) for efficiency [15, 14]. Unlike them, transkernel offloads kernel workloads from a strong processor to a weak peripheral core on the same chip.

DBT DBT has been used for system emulation [7] and binary instrumentation [34, 27, 49, 21]; DeVuyst et al. [18] uses DBT to speed up process migration. Related to transkernel, prior systems run translated user programs atop an emulated syscall interface [7, 29, 87]. Unlike them, transkernel translates kernel code and emulates a narrow interface *inside* the kernel. Prior systems use DBT to run binaries in commodity ISAs (e.g., x86) on specialized VLIW cores and hence gain efficiency [9, 35, 73, 74]. None runs on microcontrollers to our knowledge. transkernel demonstrates that DBT can gain efficiency even on off-the-shelf cores. Existing DBT engines leverage ISA similarities, e.g., between aarch32 and aarch64 [17, 16]. They still fall into the classic DBT paradigm, where the host ISA is brawny and the guest ISA is wimpy (i.e., lower register pressure). With an inverse DBT paradigm, ARK addresses very different challenges. Much work is done on optimizing DBT translation rules, using optimizers [28, 3] or machine learning[86]. Compared to them, ARK leverages ISA similarities and hence reuses code optimization already

in guest code by guest compilers.

Kernels and drivers The transkernel is inspired by the POSIX emulator [30] however is different as it emulates kernel ABIs. Prior kernel studies show rapid evolution of the Linux kernel and the interfaces between kernel layers are unstable [68, 67]. This observation motivates transkernel. Extensive work transplants device drivers to a separate core [23], user space [22], or a separate VM [39]. However, the transplant code cannot operate independent of the kernel, whereas transkernel must execute autonomously.

Encapsulating the NetBSD kernel subsystems (e.g., drivers) behind stable interfaces respected by developers, rump kernel [33] seeks to enable their reuse in foreign environments, e.g., hypervisors. The transkernel targets a different goal: spanning a live Linux kernel instance over heterogeneous processors. Applying Rump kernel's approach to Linux is difficult, as Linux intentionally rejects interface stability for drivers [36].

Suspend/resume's inefficiency raises attention for cloud servers [53, 89] and mobile [38]. Drowsy [38] mitigates inefficiency by reducing the devices involved in suspend/resume through user/kernel co-design; Xi *et al.* propose to reorder devices to resume [89]. While acknowledging the value of such kernel optimizations, we believe ARK is a key complement that works on unmodified binaries. ARK can co-exist with the mentioned optimizations in the same kernel. PowerNap [53] takes a hardware approach to speed up suspend/resume for servers. It does not treat kernel execution for operating diverse IO on embedded platforms. Kernels may put idle devices to low power at runtime [90], complementary to suspend/resume that ensures all devices are off.

9 Conclusions

We present transkernel, a new executor model for a peripheral core to execute a commodity kernel's phases, notably device suspend/resume. The transkernel executes the kernel binary through cross-ISA DBT. It translates stateful code while emulating stateless services; it picks a stable ABI for emulation; it specializes for hot paths; it exploits ISA similarities for DBT. We experimentally demonstrate that the approach is feasible and beneficial. The transkernel represents a new OS design point for harnessing heterogeneous SoCs.

Acknowledgments

The authors were supported in part by NSF Award #1619075, #1718702, and a Google Faculty Award. The authors thank the paper shepherd, Prof. Timothy Roscoe, and the anonymous reviewers for their insightful feedback. The authors thank Prof. Lin Zhong for providing a JTAG debugger. The authors are grateful to numerous video game emulators that inspired this project.

References

[1] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI), 2009.

[2] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. P. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2016.

[3] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2008.

[4] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2017.

[5] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In Proc. The European Conf. Computer Systems (EuroSys), 2015.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2009.

[7] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In Proc. USENIX Annual Technical Conference (ATC), 2005.

[8] E. Blem, J. Menon, T. Vijayaraghavan, and K. Sankaralingam. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. ACM Transactions on Computer Systems (TOCS), 33(1):3, 2015.

[9] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's First 64-bit ARM Processor. IEEE Micro, 35(2):46–55, 2015.

[10] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In Proc. USENIX Annual Technical Conference (ATC), 2010.

[11] A. L. Brown and R. J. Wysocki. Suspend-to-RAM in Linux. In Ottawa Linux Symposium, 2008.

[12] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. In Proc. ACM SIGMETRICS (SIGMETRICS), 2015.

[13] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In Proc. Ann. Int. Conf. Mobile Computing & Networking (MobiCom), 2015.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In Proc. The European Conf. Computer Systems (EuroSys), 2011.

[15] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2010.

[16] A. d'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján. HyperMAMBO-X64: Using Virtualization to Support High-Performance Transparent Binary Translation. In Proc. Int. Conf. Virtual Execution Environments (VEE), 2017.

[17] A. d'Antras, C. Gorgovan, J. Garside, and M. Luján. Low Overhead Dynamic Binary Translation on ARM. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2017.

[18] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[19] eLinux.org. PandaBoard Power Measurements. http://elinux.org/PandaBoard_Power_Measurements.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In Proc. ACM Symp. Operating Systems Principles (SOSP), 1995.

[21] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In ACM SIGARCH Computer Architecture News, 2012.

[22] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2008.

[23] B. Gerofi, A. Santogidis, D. Martinet, and Y. Ishikawa. PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems. In Proc. Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC), 2018.

[24] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7. Technical report, 2011.

[25] M. Hähnel and H. Härtig. Heterogeneity by the numbers: A study of the ODROID XU+E big.little platform. In Y. Agarwal and K. Rajamani, editors, Proc. Workshp. Power-Aware Computing and Systems (HotPower), 2014.

[26] U. Hansson. SDIO power on/off time impacts system suspend/resume time! http://connect.linaro.org/resource/sfo17/sfo17-402/, 2017.

[27] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing Binary Translation of Dynamically Generated Code. In Proc. Int. Symp. on Code Generation and Optimization (CGO), 2015.

[28] D. Hong, C. Hsu, P. Yew, J. Wu, W. Hsu, P. Liu, C. Wang, and Y. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In Proc. Int. Symp. on Code Generation and Optimization (CGO), 2012.

[29] R. J. Hookway and M. A. Herdeg. Digital FX! 32: Combining emulation and binary translation. Digital Technical Journal, 9:3–12, 1997.

[30] J. Howell, B. Parno, and J. R. Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In Proc. USENIX Annual Technical Conference (ATC), 2013.

[31] Intel. Intel SuspendResume Project. https://01.org/suspendresume, 2015.

[32] A. Kadav and M. M. Swift. Understanding Modern Device Drivers. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[33] A. Kantee and J. Cormack. Rump Kernels No OS? No Problem! Login: USENIX Magazine, 39(5), 2014.

[34] P. Kedia and S. Bansal. Fast Dynamic Binary Translation for the Kernel. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2013.

[35] A. Klaiber. The technology behind Crusoe processors. Transmeta Technical Brief, 2000.

[36] G. Kroah-Hartman. The Linux Kernel Driver Interface – Stable API Nonsense. https://www.kernel.org/doc/Documentation/

process/stable-api-nonsense.rst. (Accessed on 05/04/2019).

[37] M. Larabel. A Stable Linux Kernel API/ABI? "The Most Insane Proposal" For Linux Development. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Stable-API-ABI, 2016.

[38] M. Lentz, J. Litton, and B. Bhattacharjee. Drowsy Power Management. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2015.

[39] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2004.

[40] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In Proc. IEEE Int. Symp. on High Performance Computer Architecture (HPCA), 2010.

[41] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos. Lock-in-Pop: securing privileged operating system kernels by keeping on the beaten path. In Proc. USENIX Annual Technical Conference (ATC), 2017.

[42] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[43] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2014.

[44] R. Liu and F. X. Lin. Understanding the Characteristics of Android Wear OS. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2016.

[45] X. Liu, T. Chen, F. Qian, Z. Guo, F. X. Lin, X. Wang, and K. Chen. Characterizing Smartwatch Usage in the Wild. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017.

[46] LKML. [GIT PULL] PM updates for 2.6.33, 2009.

[47] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo. A Performance Study of Big Data on Small Nodes. Proc. VLDB Endow., 8(7):762–773, 2015.

[48] G. Lu, J. Zhan, X. Lin, C. Tan, and L. Wang. On Horizontal Decomposition of the Operating System. CoRR, abs/1604.01378, 2016.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2005.

[50] LWN. Redesigning asynchronous suspend/resume. https://lwn.net/Articles/366915/, 2009.

[51] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2013.

[52] MediaTek. Microsoft Azure Sphere MCU with extensive I/O peripheral subsystem for diverse IoT applications. https://www.mediatek.com/products/azureSphere/mt3620, 2018.

[53] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2009.

[54] D. Meisner and T. F. Wenisch. DreamWeaver: architectural support for deep sleep. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[55] Micron Technology, Inc. TN4201 LPDDR2 System Power Calculator. https://www.micron.com/support/tools-and-utilities/power-calc, 2013.

[56] Mike Turquette. The Common Clk Framework. https://www.kernel.org/doc/Documentation/clk.txt.

[57] C. Min, W. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In Proc. The European Conf. Computer Systems (EuroSys), 2018.

[58] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. IEEE Micro, 28(3):26–41, 2008.

[59] J. Morrison, D. Yang, and C. Davis. Apple watch: teardown. https://www.techinsights.com/about-techinsights/overview/blog/apple-watch-teardown/. (Accessed on 01/10/2019).

[60] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2007.

[61] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2009.

[62] NXP Semiconductors. i.MX 6SoloX - fact sheet. https://www.nxp.com/docs/en/fact-sheet/IMX6SOLOXFS.pdf. (Accessed on 05/14/2019).

[63] NXP Semiconductors. i.MX 8M Family of Applications Processors Fact Sheet. https://www.nxp.com/docs/en/fact-sheet/i.MX8M-FS.pdf. (Accessed on 05/14/2019).

[64] NXP Semiconductors. i.MX 7DS power consumption measurement. https://www.nxp.com/docs/en/application-note/AN5383.pdf, 2016.

[65] NXP Semiconductors. i.MX 7 Series Applications Processors | Arm® Cortex®-A7, Cortex-M4 | NXP. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-7-processors:IMX7-SERIES, 2017. (Accessed on 05/14/2019).

[66] H. Oi. A Case Study of Energy Efficiency on a Heterogeneous Multi-Processor. SIGMETRICS Perform. Eval. Rev., 45(2):70–72, 2017.

[67] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In J. S. Sventek and S. Hand, editors, Proc. The European Conf. Computer Systems (EuroSys), 2008.

[68] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In ACM SIGOPS Operating Systems Review, 2006.

[69] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford. Web browser workload characterization for power management on HMP platforms. In Proc. IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES), 2016.

[70] A. Ponomarenko. ABI Compliance Checker. https://lvc.github.io/abi-compliance-checker/, 2018.

[71] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2011.

[72] A. Reid. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In Proc. Formal Methods in Computer-Aided Design (FMCAD), 2016.

[73] S. Rokicki, E. Rohou, and S. Derrien. Hardware-accelerated dynamic binary translation. In Proc. ACM/IEEE Design Automation & Test in Europe Conf. (DATE), 2017.

[74] S. Rokicki, E. Rohou, and S. Derrien. Supporting runtime reconfigurable VLIWs cores through dynamic binary translation. In 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, 2018.

[75] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2018.

[76] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In Proc. Int. Conf. Ubiquitous Computing (UbiComp), 2015.

[77] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2013.

[78] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2005.

[79] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2004.

[80] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2003.

[81] Texas Instruments. AM5728 Sitara Processor: Dual Arm Cortex-A15 & Dual DSP, Multimedia | TI.com. http://www.ti.com/product/AM5728. (Accessed on 05/14/2019).

[82] Texas Instruments. Cortex-M3: Processor technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/index.html. (Accessed on 05/07/2019).

[83] Texas Instruments. OMAP4 Applications Processor: Technical Reference Manual. http://www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf, 2010. (Accessed on 05/14/2019).

[84] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. FarmBeats: An IoT Platform for Data-Driven Agriculture. In Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI), 2017.

[85] VMWARE. Virtual Machine to Physical Machine Migration. https://www.vmware.com/support/v2p/doc/V2P_TechNote.pdf, 2004.

[86] W. Wang, S. McCamant, A. Zhai, and P.-C. Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2018.

[87] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba. Enabling Cross-ISA Offloading for COTS Binaries. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2017.

[88] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev., 43(2):76–85, 2009.

[89] S. L. Xi, M. Guevara, J. Nelson, P. Pensabene, and B. C. Lee. Understanding the Critical Path in Power State Transition Latencies. In Proc. ACM/IEEE Int. Symp. Low Power Electronics & Design (ISLPED), 2013.

[90] C. Xu, F. X. Lin, Y. Wang, and L. Zhong. Automated OS-level Device Power Management for SoCs. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2015.

[91] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2013.

[92] S. Zhai, L. Guo, X. Li, and F. X. Lin. Decelerating Suspend and Resume in Operating Systems. In Proc. ACM Workshp. Mobile Computing Systems & Applications (HotMobile), 2017.

[93] Q. Zhu, M. Zhu, B. Wu, X. Shen, K. Shen, and Z. Wang. Software Engagement with Sleeping CPUs. In Proc. Workshp. Hot Topics in Operating Systems (HotOS), 2015.

Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FINELAME

Henri Maxime Demoulin Isaac Pedisich Nikos Vasilakis
Vincent Liu Boon Thau Loo Linh Thi Xuan Phan
University of Pennsylvania

Abstract

Denial of service (DoS) attacks increasingly exploit algorithmic, semantic, or implementation characteristics dormant in victim applications, often with minimal attacker resources. Practical and efficient detection of these asymmetric DoS attacks requires us to (i) catch offending requests in-flight, before they consume a critical amount of resources, (ii) remain agnostic to the application internals, such as the programming language or runtime system, and (iii) introduce low overhead in terms of both performance and programmer effort.

This paper introduces FINELAME, a language-independent framework for detecting asymmetric DoS attacks. FINELAME leverages operating system visibility across the entire software stack to instrument key resource allocation and negotiation points. It leverages recent advances in the Linux extended Berkeley Packet Filter virtual machine to attach application-level interposition probes to key request processing functions, and lightweight resource monitors—user/kernel-level probes—to key resource allocation functions. The data collected is used to train a model of resource utilization that occurs throughout the lifetime of individual requests. The model parameters are then shared with the resource monitors, which use them to catch offending requests in-flight, inline with resource allocation. We demonstrate that FINELAME can be integrated with legacy applications with minimal effort, and that it is able to detect resource abuse attacks much earlier than their intended completion time while posing low performance overheads.

1 Introduction

Denial-of-Service (DoS) attacks aim to hinder the availability of a service from its legitimate users. They work by overwhelming one or more of the resources of the service (*e.g.*, , CPU, network, memory, or disk), causing the service to become slow or, in the limit, entirely unavailable.

Classic DoS attacks are simple in structure: attackers, in large-scale, brute-force volumetric attacks, send many requests that far exceed the service's available resources. Although potentially crippling—sometimes reaching aggregate volumes of terabits per second [24, 43]—many effective mitigation techniques have been developed over the years, including commercial services like CloudFlare, Akamai, or the intrusion detection systems of Arbor Networks.

In response to these defenses, recent attacks have become much more sophisticated in nature: rather than relying on the sheer volume, they take the form of highly specialized, application-specific *asymmetric* DoS (ADoS) attacks [11, 12, 36, 48]. These attacks contain carefully-crafted, pathological payloads that target algorithmic, semantic, or implementation characteristics of the application's internals. They require significantly lower volumes of traffic and attacker resources to compromise resource availability. With the prevalence of third-part libraries, broad swaths of applications can be vulnerable to a given attack. For instance, the Regular-Expression DoS (ReDoS) attack [12, 13, 51] affects many programs that use regular expressions by leveraging algorithmic complexity to craft a single payload of a few characters that can occupy a service for several hours.

Due to this increase in sophistication, existing defenses are becoming inadequate [10, 26–28, 31, 40, 54, 60–62]. Network-based defenses are generally ineffective against ADoS attacks because these attacks lack identifiable problematic patterns at the network level. To be successful, network tools would not only need to perform deep packet inspection, but would also need to be able to predict which requests will hog resources a priori—a challenge analogous to solving the halting problem. Similarly, existing application-level defenses are limited in their efficacy: since these attacks can target arbitrary resources and arbitrary components of the service, which may be written in different programming languages and contain multiple binary third-party packages whose source code is not available or with complex dependencies, manual instrumentation of the application is prohibitively difficult, expensive, and time-consuming.

This paper presents the design and implementation of FINELAME (Fin-Lahm), a practical framework for detect-

ing ADoS attacks. In FINELAME, users only need to annotate their own code to mark the start and end of request processing; in many cases, annotations are not even required as applications lend themselves naturally to this demarcation. Our interaction with the most recent Apache Web Server[1] and Node.js[2] versions, for example, involves tracing three and seven functions, respectively, and not a single modification in their source code. Based on the annotations, FINELAME automatically tracks CPU, memory, storage, and networking usage across the entire application (even during execution of third-party compiled binaries). It does so with low overhead and at an ultra-fine granularity, which allows us to detect divergent requests before they leave the system and while they are attempting to exhaust resources.

Enabling our approach is a recent Linux feature called extended Berkeley Packet Filter (eBPF). eBPF enables the injection of verified pieces of code at designated points in the operating system (OS) and/or application, regardless of the specific programming language used. The OS is a natural, *de facto* layer of resource arbitration, with extensive infrastructure and pluggable tooling for fine-grained resource monitoring and distribution. By interposing on key OS services, such as the network stack, the scheduler, and user-level memory management facilities, FINELAME can detect abnormal behavior in a unified fashion across the entire software stack at run time.

FINELAME consists of three synergistic components that operate at the user/kernel interface. The first component allows attaching application-level interposition probes to key functions responsible for processing requests. These probes are based on inputs from the application developers, and they are responsible for bridging the gap between application-layer semantics (*e.g.*, HTTP requests) to its underlying operating system carrier (*e.g.*, process IDs). Examples of locations where those probes are attached include event handlers in a thread pool. The second component attaches resource monitors to user or kernel-space data sources. Examples of such sources include the scheduler, TCP functions responsible for sending and receiving packets on a connection, and the memory manager used by the application. To perform anomaly detection, a third component deploys a semi-supervised learning model to construct a pattern of legitimate requests from the gathered data. The model is trained in the user space, and its parameters are shared with the resource monitors throughout the system, so that anomaly detection can be performed in-line with resource allocation.

In summary, we make the following contributions:

- A novel, backward-compatible architecture at the user/kernel interface for transparently implanting resource monitors, exposed to applications via a probe API.

[1] 2.4.38 at the time of this writing

[2] v12.0.0-pre, 4a6ec3bd05e2e2d3f121e0d3dea281a6ef7fa32a on the Master branch at the time of this writing

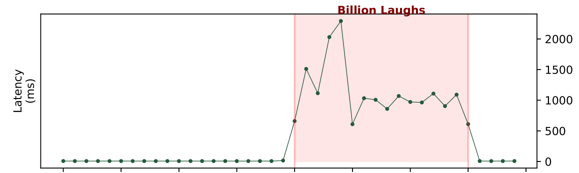


Fig. 1: Billion Laughs (XML Bomb) Attack. Under a normal load of about 500 requests per second, legitimate users experience a median of 6.75ms latency. After a short period of time, we start a malicious load of 10 requests per second (shaded area). XML bombs can take up to 200ms to compute (*vs.* a median of about 60*ns* for normal input). As a results, legitimate requests get serviced much slower, experiencing up to 2s latency. Setup details covered in (§6).

- A library of resource monitors and associated probes that can be used to detect asymmetric DoS attacks.
- An eBPF-based implementation and evaluation of FINELAME on Linux.

Our evaluation shows that FINELAME requires low additional instrumentation overhead, requiring between 4-11% additional overhead for instrumenting web applications ranging from Apache, Node.js, and DeDOS [15]. Moreover, when evaluated against real application-layer attacks such as ReDOS [5], Billion Laughs [3], and SlowLoris [46], FINELAME is able to detect the presence of these attacks in near real-time with high accuracy, based on the attack deviation from normal behavior.

The rest of the paper is structured as follows: it first motivates FINELAME's goals by providing a brief overview of asymmetric DoS attacks (§2); it then lays out our threat model and assumptions (§3); it describes the FINELAME's design and its three component parts, (i) request mapping (§4.1), (ii) resource monitoring (§4.2), and (iii) anomaly detection (§4.3); it next details several prototype implementations (§5) and evaluates the FINELAME prototypes' intrusiveness, overheads and accuracy, using a combination of micro-benchmarks and real applications (§6); finally, it compares with prior work (§7) and concludes with a discussion of limitations and possible directions for future research (§8).

2 Motivation

We begin by showing via an example server-side application the operation of an ADoS attack, the limitations of current detection mechanisms, and design goals for our system.

2.1 Background on ADoS Attacks

Fundamentally, asymmetric DoS attacks are attacks that leverage application-specific behaviors to cause disproportionate harm to the system using comparatively low amount of attacker resources. They can target any layer of the stack and

any resource within the system. ADoS vulnerabilities are widespread and often affect entire software ecosystems [41]. We detail a few of them below.

Regular-expression DoS (ReDoS) [12, 13, 51]. ReDoS attacks target programs that use regular expressions. Attackers craft patterns that result in worst-case asymptotic behavior of a matching algorithm. An example pattern is `(a+)+`, which does not match any string of the form `a*X`, but requires the system to check 2^N decomposition of the pattern to reach that conclusion, where N is the length of the target string.

XML Bomb [3]. An XML bomb (or Billion-Laughs attack) is a malicious XML document that contains layers of recursive data definitions[3], resulting in quadratic resource consumption: a 10-line XML document can easily expand to a multi-gigabyte memory representation and consume an inordinate amount of CPU time and memory on the server. Fig. 1 illustrates the impact of XML bombs on the latency of requests on a susceptible server. Under normal operation, a load of 500 legitimate requests per second are served in less than 10 milliseconds each; under a low-volume attack of 10 XML bombs per second, the latency jumps up to more than two seconds. An XML bomb affects any serialization format that can encode references (*e.g.*, YAML, but not JSON).

Improper (de-)serialization [47, 52, 53]. This class of attacks encompasses those where malicious code can be injected into running services. These vulnerabilities are, unfortunately, common in practice, and they allow malicious users to, for instance, inject a `for (;;) {}` loop to stall a process indefinitely.

Event-handler Poisoning (EHP) [14]. Attacks like the preceding can be additionally amplified in an event-driven framework. In event-handler poisoning, attackers exploit the blocking properties of event-driven frameworks so that, when a request unfairly dominates the time spent by an event handler, other clients are further blocked from proceeding. Any slowdown, whether it is in the service itself or in its recursive layers of third-party libraries can contribute to this head-of-line blocking.

2.2 Design Goals

The attacks in the previous section highlight several goals that drive FINELAME's design (§4) and implementation (§5).

In-flight Detection. Actions often need to be taken while the offending requests are "in the work"—for example, when a single request can bring the system down (*e.g.*, cooperative scheduling) or when subsequent defenses cannot be deployed (*e.g.*, IP spoofing). DoS detection needs to catch such requests *before* they leave the system, by monitoring resource consumption at a very fine temporal and spatial granularity.

[3] For example, the first layer consists of 10 elements of the second layer, each of which consists of 10 elements of the third layer, and so on.

Resource Independence. ADoS attacks may target arbitrary system-level resources (CPU, memory, storage, or networking), and may even target multiple resources (*i.e.*, multi-vector attacks). A desirable solution needs to be agnostic to the resource and able to handle any instance of inordinate consumption.

Cross-component Tracking. Given the complex structure of modern applications, ADoS attacks can also cross the boundaries of the application's internal components or processing phases. For instance, if a request causes the triggering of a timeout to an event queue, resources consumed by the initial request parsing and the timeout should both be attributed to the same request.

Language Independence. Applications today combine several ready-made libraries, which are written in multiple programming languages and often available only as compiled binaries. Thus, DoS detection should remain agnostic to the application details such as the programming language, language runtime, and broader ecosystem (*e.g.*, packages, modules).

Minimal Developer Effort. Detection needs to impose minimal burden to developers and devops, who should benefit from DoS mitigation without having to study the application internals. Rather than presenting developers with an overabundance of configuration knobs, a DoS detection system should direct precious human labor at sprinkling applications with key semantic information utilized at runtime.

3 Threat Model

To be more concrete, FINELAME assumes the following about the attacker and the broader environment.

Threats. We consider a powerful remote attacker that (i) can send arbitrary requests to a service hosting a vulnerable application, (ii) has control over potentially all of the application's legitimate clients, and (iii) is aware of the application's structure and vulnerabilities, including exploits in its dependency tree. We do not distinguish between legitimate and malicious clients who intersperse harmful requests that attack resources with one or more benign requests. Specifically, any subset of hosts can send any number of requests that may or may not attack any subset of resources. We do not limit resources of interest to CPU; attackers can target memory, file descriptors, or any other limited resource in the host system. That means that attacks can take the form of a single client attempting to consume 100% of the CPU indefinitely, or of multiple attacks from multiple clients over many of the system's resources.

Assumptions. We assume (i) vulnerable but not actively malicious code, and (ii) that FINELAME sees at least some benign traffic. If all traffic is malicious from the beginning, in-flight detection and mitigation become less urgent, as anomalies become the norm, and the application owners should first

revise their deployment pipeline. We also assume that the resource utilization of request processing can be attributed to a single request by the end of each processing phase, even if the processing phases is split into multiple phases across different application components. As keeping a reference to the originating request is a natural design pattern, in all of the services we tested, a unique identifier was already available; in cases where there is no such identifier, one must be added, and we detail how to do so in section 4.

4 FINELAME Design

Figure 2 depicts the overall design of FINELAME. Conceptually, FINELAME consists of three main components:

- *Programmer annotations* that mark when a request is being processed. FINELAME requires only a few annotations, even for complex applications, to properly attribute resource utilization to requests.
- *Fine-grained resource monitors* that track the resource utilization of in-flight requests at the granularity of context switches, mallocs, page faults.
- A *cross-layer anomaly detection* model that learns the legitimate behavior and detects attacks as soon as they deviate from such behavior.

Programmers can use FINELAME by annotating their application with what we call *request-mappers*. These annotations delineate, for each component and processing phase, the start and end of processing, as well as the request to which resource utilization should be attributed. For example, in an event-driven framework, the beginning and the end of each iteration of the event handler loop should be marked as the start and the end of a request's processing, respectively.

At runtime, when FINELAME is installed on the host environment, FINELAME attaches small, low-overhead *resource monitors* to particular points in the application or operating system. The aforementioned request-mappers enable FINELAME to determine the request to which the resource consumed by a thread or process should be credited. In section 5, we detail our out-of-the-box FINELAME library of request-mappers and resource monitors for several popular cloud frameworks. Our library tracks the utilization of a range of key OS-level resources; however, programmers can further extend it with user-level resource monitors to track application-specific resources (*e.g.*, the occupancy of a hash table).

Finally, FINELAME's monitoring data is used to perform lightweight, inline anomaly detection. Resource monitors first feed data to a machine learning model training framework that computes a fingerprint of legitimate behavior. Parameters of the trained model are installed directly into the resource monitors, which evaluate an approximation of the model to automatically detect anomalous behavior on-the-fly. The end result of FINELAME is a system for high-accuracy, fine-grained, and general ADoS attack detection.

4.1 Request-mapping in FINELAME

Conceptually, there are three operations in request mapping:

- `startProcessing()`: This annotation denotes the start of a processing phase. Any resource utilization or allocations after this point are attributed to a new unique request.
- `attributeRequest(reqId)`: As soon as we can determine a unique and consistent request identifier, we map the current processing phase to that request. For instance, when reading packets from a queue, if the best consistent identifier for a packet is its 5-tuple, resource tracking would start as soon as the packet is dequeued, but would only be attributed to a consistent request ID after Layer-3 and Layer-4 processing are completed. In general, `attributeRequest(reqId)` is called directly after `startProcessing()`, and depending on the specific of the application, the two can sometimes be merged (§ 5).
- `endProcessing()`: Finally, this operation denotes the completion of processing, indicating that subsequent utilization should not be attributed to the current request.

In order for the resource monitors to properly attribute utilization to a request, FINELAME requires programmers to annotate their applications using the above three *request mapping* operations. Ideally, the annotations should cover as much of the code base as possible; however, not all resource utilization can be attributed to a single request. In such cases, programmers have flexibility in how they perform mapping: for true application overhead—rather than request processing overhead—utilization can remain unattributed, and for shared overhead (*e.g.*, garbage collection), utilization can be partitioned or otherwise assigned stochastically.

Every request is given an identifier that must be both unique and consistent across application components and processing phases. This identifier is used to maintain an internal mapping between OS entity (process or thread) and the request. Example identifiers include the address of the object representing the request in the application, a request ID generated by some application-level tracing solution [7, 20, 29, 34, 45, 49, 55], or a location in memory if the request is only processed once. From the moment a `startProcessing` annotation is called to the moment the `endProcessing` annotation is called, FINELAME will associate all the resources consumed by the OS entity to the request.

An optimization of this technique can be implemented when the application lends itself naturally to such mapping

Fig. 2: FINELAME overview. Key elements: (1, right) user and kernel data-collection probes at points where an HTTP request interacts with resource allocation; (2, mid-left) a data structure shared between user and kernel space, that aggregates and arranges collected data; (3, left) a userspace training component that instantiates model parameters, fed back to the probes. Information flow between 1–3 is bidirectional.

between OS entity and request. For instance, event-driven frameworks or thread-pool based services usually have a single or small number of entry points for the request, to which FINELAME can readily attach request-mappers via eBPF without source code modification. We found this optimization to be the common case, and FINELAME does not require any modification to the application we explore in section 6.

4.2 Resource Monitoring in FINELAME

Resource tracking between `startProcessing` and `endProcessing` annotations are done via a recent Linux kernel feature called eBPF. We first provide some background on the operation of eBPF, and then discuss how we utilize it to perform extremely fine-grained resource monitoring of in-flight requests.

4.2.1 Background on eBPF

The original Berkeley Packet Filter (BPF) [35] has been a long-time component of the Linux kernel networking subsystem. It is a virtual machine interpreting a simple language traditionally used for filtering data generated by kernel events. Notable use cases are network packets parsing with Tcpdump [56] and filtering access to system calls in the seccomp facility. In version 3.0 a just-in-time compiler was implemented, allowing for a considerable speedup of the processing of BPF programs by optimizing them on the fly.

In version 3.15, Alexei Starovoivtov significantly extended BPF (dubbing the new system “eBPF”). The new version has access to more registers and an instruction set mimicking a native RISC ISA, can call a restricted subset of kernel functions, and can share data from kernel-space to user-space through hash-like data structures. While eBPF is a low-level language, users can write programs in higher languages such as C (and even Python with the BCC project [2]) and generate eBPF code with compilers such as GCC and LLVM.

Generated programs are verified before being accepted in the kernel. The verifier imposes a set of strict constraints to eBPF programs to guarantee the safety of the kernel. Common constraints include the absence of floating point instructions, a limit of 4096 instructions per program, a stack size capped at 512 Bytes, no signed division, and the interdiction of back-edges in the program’s control flow graph (*i.e.*, no loops).

The ability of eBPF programs to be attached to both kernel and user-space functions and events, their extremely low overhead, and their ability to share data with user space without the need for any IPC or queuing mechanism make eBPF a prime candidate for implementing resource monitors in FINELAME.

4.2.2 Resource Monitor Architecture

FINELAME’s resource monitors are attached to various user- and kernel-space data sources (*e.g.*, the scheduler or TCP stack) and use the mapping described in section 4.1 to associate resource consumption to application-level workflow (*e.g.*, HTTP requests). A resource monitor requires the following information: the type and name of the data source, and potentially the path of its binary.

Our current prototype of FINELAME uses the features listed in Table 1. When executed, most resource monitors operate under the following sequence of actions: i) verify whether a request mapping is active for the current PID and exit if not; ii) collect the metric of interest (usually through the arguments of the function triggering it) and store it, time-stamped, in a shared data structure; and iii) perform anomaly detection on the request if the model’s parameters are available (see section 4.3).

The time a request spends executing instructions on a processor is represented by *cputime*. We instrument both the *scheduler_tick()* and the *finish_task_switch()* kernel functions, which are called at every timer interrupt and context switch, respectively, to either start a timer when a thread executing a registered request is scheduled for execution or collect the amount of CPU time consumed by the task swapped out. We instrument the *tcp_sendmsg()* and *tcp_rcleanbuf()* to collect *tcp_sent* and *tcp_rcvd*, the amounts of bytes sent and read from a TCP connection, respectively. To compute *tcp_idle_time*, which represents the period of inactivity from the sender on a TCP connection, we measure the time elapsed between two occurrences of *tcp_cleanup_rbuf()*. To monitor the heap memory consumption occasioned by the processing

Name	Description	Event	Type
tcp_idle_time	Inactivity time on a TCP connection	tcp_cleanup_rbuf	kernel probe
tcp_sent	Bytes sent through TCP connections	tcp_sendmsg	kernel probe
tcp_rcvd	Bytes received through TCP connections	tcp_cleanup_rbuf	kernel probe
cputime	Amount of CPU time consumed	scheduler_tick, finish_task_switch	kernel probe
malloc_memory	Bytes allocated through the *malloc* function	glibc_malloc	user probe
page_faults	Number of page faults events	exceptions:page_fault_user	kernel tracepoint

Tab. 1: Default resource monitors in FineLame.

of a request, we monitor the glibc *malloc* function. Applications where memory management is partly handled by the runtime (such as in Python) can be monitored in a similar fashion. Likewise, the model can be generalized to garbage collected languages. Finally, we monitor the page fault events in the application by attaching a resource monitor to the *exception: page_fault_user* kernel tracepoint. We observed in our evaluation that CPU time was the best discriminant for CPU based attacks, while connection idle time the best for slow attacks (such as Slowloris and RUDY).

The above default, general-purpose resource monitors in FineLame are sufficient for a large set of existing applications; however, it can be extended to all the kernel events available for tracing and probing, as well as user-level functions (to monitor application-level metrics). If any application-level metrics are required (such as data structure occupancy, counters, and so on), programmers can augment our resource monitors with custom eBPF programs attached to arbitrary probe points in either kernel- or user-space.

4.3 Attack Detection in FineLame

Detection algorithm. For fast detection, FineLame is designed to enable anomaly detection as close as possible to the resource allocation mechanism. Without a method for in-flight anomaly detection *in addition to* mechanisms for in-flight resource tracing, detection and mitigation of in-flight requests would not be possible.

This detection problem can be reduced to quantizing the abnormality of a vector in *n*-dimensional space. Once a sufficient amount of data has been gathered to compute a fingerprint of the legitimate requests' behavior, we can train an anomaly detection model. The model can span all the metrics collected by the resource monitors, allowing us to detect abuse on any of the resources of the system as well as cross-resource (multi-vector) attacks.

For the unsupervised version of this problem, the most popular methods take one of two approaches: distance-based or prediction-based. The former family of models aims to cluster known, legitimate data points and compute the distance of new data points to those clusters—distance that is used to quantify the anomaly. The latter family assumes the existence of a set of input data points that are correct, and learns a function representing those points. When a new point enters the system, the model computes the value of the learned function; the prediction error is then used to quantify the degree of

```
Required       | FPAS         # FPA scaling factor
data           | pid_to_rid   # OS carrier to request
structures     | req_points   # Request profiles
               | model_params # K-means parameters
               | dp_dists     # Distances to centroids
               | thresholds   # Alerts cut-off bar
               |
               | fun resource_monitor(context):
Is there a     |  pid = bpf_get_current_pid()
mapping?       |  rid = pid_to_rid.get(pid)
               |  if (rid):
               |     ts = get_timestamp()
               |     metric = context.get_arguments()
               |     dp = req_points.get(rid)
Update         |     if (dp):
request        |        dp.update(metric, ts)
profile        |     else:
               |        dp = init_dp(rid, metric, ts)
               |        req_points.insert(dp)
               |     μ, σ = model_params.get()
If anomaly     |     if (μ && σ):
detection      |        metric_scaled = metric << FPAS
parameters     |        metric_scaled -= μ
are available, |        if metric_scaled < 0:
scale and      |            metric_scaled *= -1
standardize    |            metric_scaled /= σ
the data in    |            metric_scaled *= -1
the FPA        |        else:
space          |            metric_scaled /= σ
               |        min_dist, closest_k
               |        #pragma loop unroll
               |        for k in K:
               |            current_dist =
               |              dp_dists.get(dp, k)
               |            new_dist =
Update         |              metric_scaled+current_dist
distance to    |            dp_dists.update(dp, new_dist)
clusters       |            if (new_dist < min_dist):
               |                min_dist = new_dist
               |                closest_k = k
Perform        |        t = thresholds.get(closest_k)
anomaly        |        if new_dist > t:
detection      |            report(rid, dp, s)
```

Fig. 3: FineLame anomaly detection. Pseudocode for FineLame's inline anomaly detection.

anomaly.

Because of the training complexity, prediction complexity, and required training data, many existing solutions in both distance-based and prediction-based categories are impractical to execute at fine granularity. For instance, the popular algorithm DBSCAN [18] is not suitable for FINELAME, as it requires us to evaluate the distance of new data points to all the possible "core" data points in the model. The amount of data points considered (and therefore the size of the model) is usually linearly proportional to the size of the training set. Some accurate approximations of DBSCAN have been proposed [22], but even with a small number of clusters, almost all of the training dataset still needs to be part of the model. Likewise, the performance of prediction-based models made on neural networks, such as Kitsune [38], is highly dependent on the depth and width of the model. The amount of parameters of such networks grows exponentially with the number and size of the hidden layers.

Given the above concerns, we chose to implement anomaly detection in FINELAME with K-means, a technique that allows us to summarize the fingerprint of legitimate requests with a small amount of data. In K-means, the objective function seeks to minimize the distance between points in each cluster. The model parameters are then the centroids and distribution of the trained clusters. In a typical use-case scenario, FINELAME is configured to perform only request monitoring for a certain amount of time, after which it trains k-means on the monitoring data gathered in user-space from the resource monitors shared maps. In practice, we found that a K value equal to the number of request types in the application yields a reasonable estimation of the different behaviors adopted by legitimate requests, while being a number low enough such as to contain FINELAME's overhead.

Model training and deployment. Gathering the training data is done by a simple look-up from the user-space agent to the shared eBPF maps holding the requests resource consumption data. Using those profiles, the user-space agent standardizes the data (center to 0 and cast to unit standard deviation). Subsequently, the agent trains K-means to generate a set of centroids representing the fingerprint of the good traffic. The parameters of the model, to be shared with the performance monitors, are then the cluster centroids, as well as the mean μ and standard deviation σ of each feature in the dataset, and a threshold value τ statistically determined for each cluster.

As described above, the performance monitors have limited computing abilities and do not have access to floating point instructions. Thus, they are designed to perform fixed point arithmetic in a configurable shifted space, and require FINELAME's to shift the model parameters in this space before sharing them. Using two precision parameters a and b, each datapoint is transposed in a higher space 10^a, and normalized such that the resulting value lies in an intermediate space 10^{a-b}, retaining a precision of $a-b$ digits. This means that during the normalization operation each parameter value x undergoes the following transformation: $x = \frac{(x*10^a)-(\mu*10^a)}{\sigma*10^b}$.

Application	Request mapping probes	SLOC
Apache	5	41
Node.js	9	64
DeDoS	2	21

Tab. 2: Intrusiveness of FINELAME, quantified.

Once standardized, the clusters' centroids as well as each feature's mean and standard deviation are shared with the resource monitors through eBPF maps. Upon availability of those parameters, the resource monitors update not only the resource consumption of existing requests, but also their *outlier scores*, a measure we use to quantify the degree of anomaly of a request. Due to the constraints imposed on eBPF programs—specifically, taking a square root is complex as we do not have access to loops—we choose the normalized L1 distance to the closest cluster as the outlier score. While being a crude measure, the L1 is equivalent to more complex norms as resource vectors are of finite dimension. It preserves information about which resource is abused, and it lets us set statistical thresholds to determine cut-off points used for flagging abnormal requests. The algorithm for this entire process is shown in Figure 3.

Finally, we note that because FINELAME is primarily designed toward the detection of resource exhaustion attacks, we allow the anomaly detection engine to maintain signed values for outlier scores. This means that requests that have not reached their expected legitimate amounts of resource consumption, and that would look abnormal in an absolute value setting, are not flagged as such. This is important because it highlights the fact that FINELAME is not geared toward volumetric attacks that aim to bring the system down with a vast amount of low consumption requests.

5 Use Cases and Implementation

To demonstrate the generality of FINELAME and the minimal developer effort required to use it, we apply FINELAME to three web platforms: Apache [1], which is estimated to serve ~40% of all active webpages; Node.js [4] a popular server-side JavaScript-based web server; and DeDoS [15] an open source component-based framework for building web services. Our prototype of FINELAME is available on https://github.com/maxdml/Finelame. Table 2 quantifies the programming effort required to write request-mappers for those three applications to use FINELAME.

Apache web server. Primarily written in C, Apache's request processing is implemented by Multi-Processing Modules (MPM). In the latest versions of Apache (2.x), requests are served by multiple processes which can have multiple

worker threads themselves; each thread handles one connection at a time.

When a request enters the system, an application-level (`conn`) object is created by the `core_create_conn` function to contain it before the request is dispatched to a worker thread. Subsequently, the request is processed by either the `ap_process_http_sync_connection` or the `ap_process_http_async_connection` functions, which take the `conn` object as argument. From FINELAME, we attach one request-mapper to `core_create_conn`, and two requests-mappers to the http processing functions, one over a *uprobe* called upon entering the function, the other over a *uretprobe* called when returning from it. We exploit the `conn` object to generate a unique identifier for each request and map it to the underlying thread worker, so that resource monitors can later gather resource consumption data on the request's behalf. The mapping is undone when the function returns and the request exits the system. When a worker thread executes a new request, the request-mapper updates the mapping with the new request's ID. This solution requires no modification to the Apache source code, and 41 lines of eBPF code over 5 probes.

Node.js required more slightly more instrumentation due to its asynchronous model, which offloads work to a worker pool (implemented with *libuv* [30]). The instrumentation required eBPF probes to be attached to seven user-space functions within the libuv library. As in Apache, we found a data structure—`struct uv_stream_t`—that could (i) be used to generate a unique identifier, and (ii) was carried consistently across the disparate components of the framework.

Request-mappers were applied to the seven libuv functions as follows:

- `uv_accept`: a new request is initialized, and is associated with the `uv_stream_t` structure that handled communication with the client.
- `uv__read` and `uv__write`: the request associated with the client's stream is assigned to the current thread for the duration of the function.
- `uv__work_submit`: the request assigned to the current thread is associated with a work-request submitted to the worker pool.
- `uv__fs_work`, and `uv__fs_done`: the request associated with the work-request is assigned to the current (worker) thread.
- `uv_async_send`: the request is unassigned from the current thread.

Again, this solution requires no changes in Node.js source code, only knowledge of which functions are processing requests. The request-mappers totalized 64 lines of eBPF code.

DeDoS is an event-driven framework where programmers write and deploy their application as software components that are automatically allocated and deallocated based on demand. Each of those components monitor a local event-queue from which new requests are consumed. Unifying the disparate components is a generic event-handling function (`receive()`). Programmers implement their component's functionality inside this event-handling function.

DeDOS provides request tracing and explicitly tracks the passing of requests between components. We chose DeDoS as a proof-of-concept proxy for micro-service, event-driven applications providing request tracing capability. In these types of applications, annotation is simple as FINELAME can maintain a direct mapping between the application-level unique request identifier and the event handler's thread PID in order to track resource consumption across component boundaries. FINELAME traces only the `receive()` function class with request mappers, and does not require modifications to the framework. The request-mappers require 21 lines of eBPF code.

6 Evaluation

In this section, we present our evaluation results of FINELAME. Our evaluation is centered around the following aspects of the system:

- **Overhead.** The overhead of FINELAME compared to no monitoring, or in-application instrumentation
- **Accuracy.** The ability of FINELAME to accurately detect real attacks never seen yet by the application

6.1 Experimental setup

We present the setup on which we evaluate both the overhead and accuracy aspects of FINELAME. In all cases, the server applications are running on a 12 cores Xeon Silver 4114 at 2.20GHz , while our legitimate and attack clients are running on an Intel Xeon E5-2630L v3 at 1.80GHz. Both server and client machines have a total of 62G of RAM, and have hyper-threading and DVFS disabled.

We use version 2.4.38 of Apache, and configure it to use 50 worker threads. We use version $12.0.0 - pre$ of Node.js with the default configuration of 4 worker threads for libuv. Both Apache and Node.js are configured to serve a set of Wikipedia [59] pages. Node.js parses a regular expression provided in the request's URI to find the path of the file to serve. It's parser, *liburi*, is vulnerable to the ReDoS attack. All the applications impose a timeout of 20 seconds on connections. We deploy a simple webserver in DeDoS which can process three types of requests: serve a Wikipedia article, process a randomly generated XML file uploaded in a POST request, and parse a regular expression. The server is decomposed into

several software components: socket reading, HTTP parsing, file serving, XML parsing, regular expression parsing, and response writing. The XML parser is implemented with *libxml2*, which is vulnerable to the Billion Laughs attack.

Our good traffic is generated by Tsung [6] and explores evenly all the servers' exposed endpoints; bad traffic is generated by an in-house **C** client for the ReDoS and Billion Laughs attacks, and pylorys [23] for the Slowloris attack. Tsung generates load under an exponential distribution centered on a configurable mean, while our attack client is configured to send a fixed load.

6.2 Overhead of FINELAME

Figures 4 presents the overheads incurred by FINELAME's instrumentation on Apache, Node.js and DeDoS. In all of our experimental setups, we evaluate the legitimate client latency experienced when the server is not instrumented, when it is instrumented by FINELAME, and when FINELAME's resource monitors are also performing anomaly detection (FINELAME +). The load is as described earlier in sec 6.1, and explore all the instrumented paths in the applications. We also evaluate the cost of instrumenting the DeDoS framework itself to evaluate FINELAME overheads compared to a traditional user-space solution. The bars plot the median of the clients latency, and all our experiments are run thrice for a period of 100 seconds. In the case of Node.js the instrumentation cost adds 8.55% overheads and adding anomaly detection 9.21%. In the case of Apache, FINELAME adds 11.38% and 11.72% overheads respectively. In the case of DeDoS, the baseline latency is higher than with the two previous services, due to the fact that the application is not only serving files but also parsing POST requests, and also the framework is less optimized than the two battle-tested Apache and Node.js. Instrumenting directly the framework comes with an overhead of 2.9%, while FINELAME comes with 4.23% overheads, 6.3% if also performing anomaly detection.

In general we observe that the overheads incurred by FINELAME are higher when the baseline processing time of the service is low, and does not grow linearly with the complexity of the application. In addition, we found that performing anomaly detection in addition to monitoring resource consumption almost comes for free.

6.3 Performance of FINELAME

https://www.overleaf.com/project/5c22751775031d099f528e64 Our performance evaluation of FINELAME is centered around its ability to detect attacks requests before they exit the system, while providing accuracy competitive with non-approximated user-level algorithms.

6.3.1 Attacks

Our experiments aim to quantify the impact of attacks on quality of service. Consequently, we tune attacks strength such that they will not bring down the server but rather degrade the quality of service provided to legitimate users.

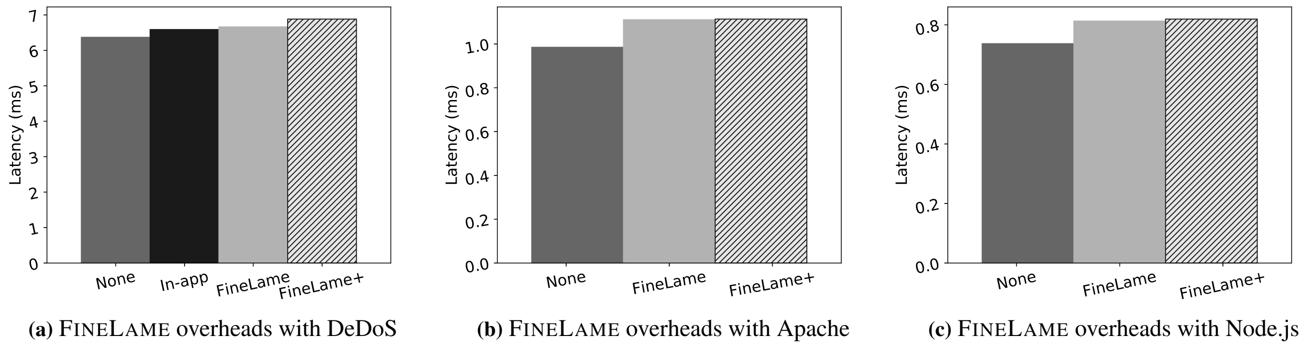
ReDoS: This attack consist of specially crafted regular expressions which are sent to the server for processing. The strength of the attack grows exponentially with the number of malicious characters present in the expression. Because the application processing units are busy handling those requests, legitimate requests get queued for a longer period of time, and ends-up being responded to more slowly.

Billion Laughs: The attack consists of XML files filled with several levels of nested entities. The parsing cost is exponentially proportional to the depth of the document. The impact is similar to the ReDoS attack.

SlowLoris: The attack consists in maintaining open connections to the server, keeping them alive by sending individual HTTP headers at a regular interval smaller than the server's timeout, but never completing the request—we assume that the attacker is able to probe the service and discover this timeout. As a result, the server's connection pool gets exhausted, and it can't answer new requests. This technique can also implement a dormant attack which cripples the ability of the server to handle surges of legitimate traffic, by denying a fraction of the total connection pool.

6.3.2 Anomaly Detection Performance

Evaluation metrics As is common with anomaly detectors, the output of FINELAME is a score which quantifies the abnormality of a request. This score is then either used as a raw metric for mitigation algorithms, or compared against a threshold τ to be transformed into a binary variable where 0 means negative (no anomaly), and 1 means positive (attack). With τ set, and using our knowledge of the ground truth, we can determine the accuracy of each of the detector's outputs as true/false positive/negative. The choice of τ is crucial, as too low a value can result in a large amount of false positive, while too high a value can induce a large amount of false negative. For our experiments, we set τ to be the outermost point for each cluster in the training set, *i.e.*, the most consuming legitimate request we've seen so far for the cluster. The challenge associated with deriving a large τ from the training traffic is that attacks can now take longer to detect—and might not be detected at all if they are too weak. This latter case does not concern us, because to bring down the system with weaker attacks, an attacker would be forced to change its method from asymmetric to volumetric. The benefit of a higher τ is that it helps decreasing the False Positive Rate (FPR, $\frac{FP}{FP+TN}$), a desirable behavior for operators using the system. For our experiments, we present the True Positive



(a) FINELAME overheads with DeDoS **(b)** FINELAME overheads with Apache **(c)** FINELAME overheads with Node.js

Fig. 4: Overhead of FINELAME with various applications

Rate (TPR, $\frac{TP}{TP+FN}$), True Negative Rate (TNR, $\frac{TN}{TN+FP}$) and F1 ($\frac{2TP}{2TP+FP+FN}$). TPR evaluates the system's ability to detect all the attack requests. TNR evaluates its ability to evaluate legitimate requests as such. The F1 score is the harmonic mean of the TPR and the recall. It evaluates both the TPR and the precision of the system.

In addition to its post-hoc instrumentation abilities and low programmer burden, the main contribution of FINELAME is it's detection pace. We evaluate the *Detection Speedup* (DS) of the system, which we define as being the delta between the time of last detection and the time to first detection, over the lifetime of the request. We expect DS to increase as users set more strict thresholds (lower values of τ), but found that even with τ set to the outermost point in each training cluster, FINELAME is able to detect attacks up to more than 97% faster.

Experiments All our experiments are run for a duration of 400 seconds, split into 3 phases. The first phase sees only legitimate traffic flowing through our target applications, and last 200 seconds. FINELAME is configured to only have the performance monitors gather data for the first 180 seconds, after which point it triggers the training of the anomaly detection model and share its parameters. Attacks start at time 200, and last for 150 seconds. A final period of 50 seconds sees the attack stop, and only good traffic is sent to the application. We perform two CPU exhaustion attacks, Billion Laughs and ReDoS, as well as a connection pool exhaustion attack, SlowLoris. For all experiments, we compare the TPR and TNR of FINELAME to a non approximated user-space implementation of K-means (that is, with floating point arithmetic) to confirm that the system is competitive with more complex user space solutions. We set $K = 3$, the maximum number of request types that the application we setup can accept, and use $a = 10$ and $b = 6$ factor to retain 4 digits in fixed point arithmetic.

Table 3 presents the detection speed and performance of FINELAME.

ReDoS: In our first experiment, we attack Node.js with three strengths of ReDoS requests. In the two first experiments, the workload is made of 98% of benign requests and 2% of malicious regular expressions blocking the event loop of the server (about 500 and 10 r/s, respectively). In the third experiment, with the strongest attack, we reduce the attack rate to 1 r/s, such that the attack does not bring down the server. Legitimate requests are served in about 0.8ms on average under normal conditions, but get delayed in proportion of the intensity of the ReDoS requests when the attack starts. During the first attack, bad requests are served in 23ms on average, a 28.75× increase compared to normal requests. Good requests are also penalized and are served in about 4ms. During the second attack, bad requests are served in 45.6ms on average, a 57× increase compared to normal requests. Legitimate requests are affected and incur an average latency of 13.5ms. During the third attack, bad requests are served in 90.9ms on average, a 113.6× increase. Legitimate requests incur an average latency of 6ms. Due to its ability to credit requests' resource consumption at the granularity of context switches, in both experiments, FINELAME is able to detect attack requests before they exist the system, at least 80.9% earlier for 50% of the bad traffic, and up to 95.3% earlier. The user-space, non-approximated evaluation of k-means using the L2 norm for measuring distances, perform only marginally better.

Billion Laughs: In this experiment, we attack DeDoS with two different strengths of Billion Laughs (XML bomb) requests. The good traffic follows a diurnal pattern, oscillating between 250 and 750 requests per second. Under normal conditions, legitimate requests are served in 6.87ms on average. In the first experiment, we send 15 malicious requests per seconds (about 2% of the peak legitimate traffic, and 6% of the lower phase), which are served in 29.28ms on average, a 4.26× increase in response time. In the second experiment, we decrease the number of bad requests to one per second (about 0.1% and 0.4% of the peak and low traffic, respectively), and increase their intensity such that they are served in 203ms in average (an order of magnitude increase compared to the first case), which represents a 29.55× increase

Attack	Strength	TPR		TNR		F1		DS		
		FL	K-means L2	FL	K-means L2	FL	K-means L2	median	75th	max
ReDoS	28.7×	100%	100%	99.995%	99.999%	99.88%	99.98%	80.9%	81.2%	83.2%
	57×	100%	100%	99.993%	99.994%	99.81%	99.83%	90.4%	90.5%	91.0%
	113.7×	100%	100%	99.997%	99.999%	99.29%	99.76%	90.9%	95.1%	95.3%
Billion Laughs	4.7×	100%	100%	100%	100%	100%	100%	83.1%	85.5%	87.7%
	34.8×	100%	100%	99.998%	99.998%	99.53%	99.76%	97.0%	97.1%	98.2%
SlowLoris	5 sockets	100%	100%	100%	100%	100%	100%	75%	n/a	n/a

Tab. 3: FINELAME TPR, and detection Speedup for Apache, Node.js and DeDoS.

in load compared to legitimate requests in normal conditions. For the weaker attack, FINELAME is able to detect malicious requests 78.83% faster than the user-space solution, at least 50% of the time, and up to and 97% faster for the strongest attack.

SlowLoris: In this experiment, we configure Apache to handle requests with 25 worker threads, and timeout on reading HTTP headers after 20 seconds. We configure the attack client to maintain 5 connections to the server opened at all times, refreshing it every 5 seconds. Effectively, this drives the *tcp_idle_time* of the malicious request high and makes them standout from the legitimate ones. This attack is "all or nothing", in the sense that it will not impact the legitimate requests until the connection pool gets exhausted. FINELAME's is able to detect the abnormal idle time about 75% faster than the application ($1 - \frac{5}{20} * 100$), which would have otherwise to experience the timeout before reporting the request.

7 Related Work

Volumetric Attack Detection There is a large body of work addressing volumetric DoS attacks [10, 26, 31, 40, 60–62], including attacks that target the network [27, 28, 54]. As described earlier (§1), these systems do not protect against *asymmetric* DoS attacks, a concern shared by both industry [32, 50] and academia [13, 14, 51].

Application-based Detection Prior works on application-layer DoS detection either depend heavily on repeated outliers, or are often deeply tied to a specific application. Techniques include comparing the entropy of offending and legitimate traffic [39, 63], sampling traffic flows [25], and sketch-based feature-dimensionality reduction [58]. While these techniques work well for volumetric attacks, they have self-assumed limitations when the attack traffic is low—the primary focus of this paper.

DSHIELD [44] is a system that assigns "suspicion scores" to user sessions based on their distance from legitimate session. While similar in nature to FINELAME's anomaly detection technique, it relies on the operator knowing all the possible classes of requests that the server can process. FINELAME anomaly detection engine learns on legitimate requests so that it does not depend on *a priori* knowledge of execution paths or vulnerabilities.

BreakApp [57] is a module-level compartmentalization system that attempts to defend against DoS attacks, among other threats stemming from third-party modules. While BreakApp's capabilities increase with more and smaller modules, FINELAME works even with monolithic applications entirely developed as a single module. BreakApp's mitigation uses simple queue metrics (*i.e.*, queue length at the module boundary *vs.* replica budget), whose cut-off parameters are statically provided by the programmer; FINELAME uses a more advanced learning model, which parameters are adjusted at runtime.

Rampart [36] focuses on asymmetric application-level CPU attacks in the context of PHP. It estimates the distribution of a PHP application function's CPU consumption, and periodically evaluates running requests to assess the likelihood they are malicious. It then builds filters to probabilistically drop offenders—repeated offenders increase their probability of being filtered out. While FINELAME profiles legitimate requests resource consumption, it is not limited to CPU-based attacks. It also works with applications with components built with many different languages.

In-kernel Detection Recent work has shown good results for mitigating ADoS attacks by exploiting low level system metrics. Radmin [16] and its successor Cogo [17] train Probabilistic Finite Automatas (PFAs) offline for each resource of a process they want to monitor, then perform anomaly detection by evaluating how likely the process' transition in the resource space is. Training the PFAs requires days in Radmin, and minutes in Cogo, while FINELAME can train accurate models in seconds or hundreds of microseconds. We expect this capability to be helpful in production systems where the model has to be updated, *e.g.*, to account for changes in an application's component. In addition, Cogo reports detection time in the order of seconds, while FINELAME's inline detection operates at the scale of the request's complexity—milliseconds in our experiments. Lastly, Radmin/Cogo operate at the granularity of processes/connections. FINELAME assumes a worst-case threat model where malicious requests are sent sporadically

by compromised clients, and thus operate at this granularity. Per-request detection has the added benefit to enable precise root cause analysis, further enhancing the practicality of FINELAME.

Programmer Annotations Prior work proposes an annotation toolkit that programmers can use in their code to specify resource consumption requirements [42]. The framework detects connections that violate the provided specification (and then attempts to mitigate by rate limiting or dropping them). Unfortunately, it requires knowledge of the application internals. Worse even, it expects developers to understand the program's expected resource consumption quite accurately. Moreover, such a hard cut does not distinguish between occasional consumption that is slightly above limits and true attackers.

Prevention-as-a-Service A recent vein of work proposed "Attack prevention as a Service", where security appliances are automatically provisioned at strategic locations in the network [19, 37]. Those techniques are largely dependent on attack detection (to which they do not provide a solution), and thus are orthogonal to our platform, which operates directly at the victim's endpoint.

Performance anomaly detection ADoS attacks are a subset of the broader topic of performance degradation, a topic that has been extensively studied. Magpie [9] instruments an application to collect events from the entire stack and obtain request profiles *post-mortem*. X-trace [21] is a tracing framework that preserves causal relationship between events, and allow the offline reconstruction of request trees. X-ray [8] builds on taint-tracking to provide record and replay system to summarize the performance of application events offline. One of FINELAME's key difference with those systems is its lightweight in-flight profiling technique, which allows us to perform anomaly detection while the request is still in the system. Retro [33] provides a tracing architecture for multi-tenant systems that enables the implementation of resource management policies. While its architecture is similar to FINELAME's, its focus is on performance degradation caused by competing workloads, rather than the detection of degradation within a single application.

While the impact can be similar, we note that for ADoS attacks, in-flight request tracking is critical to timely detection and mitigation.

8 Conclusion

In this paper, we describe and evaluate FINELAME, a novel fine-grained application-level DoS detection framework. FINELAME is designed for interaction with modern distributed applications, operates orders of magnitude faster than previous techniques, and is able to detect yet-unseen attacks on an application. FINELAME is enabled by recent advances in the Linux kernel, and bridges the gap between application-layer semantic and low-level resource allocation sub-systems. It is a first step toward deploying complex machine learning applications for fine grained services, in an era where the size of services is shrinking (micro/pico-services).

9 Acknowledgments

We would like to thank our shepherd, Mike Reiter, and the anonymous ATC reviewers for their useful feedback. This material is based upon work supported in parts by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-16-C-0056 and No. HR001117C0047, and NSF grants CNS-1513687, CNS-1513679, CNS-1563873, CNS-1703936 and CNS-1750158. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

References

[1] Apache HTTP server project. https://httpd.apache.org/.

[2] bcc on GitHub. https://github.com/iovisor/bcc.

[3] Common vulnerabilities and exposures (see cve-2003-1564). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564.

[4] Node.js server project. https://nodejs.org/en/.

[5] Regular expression denial of service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[6] Tsung. http://tsung.erlang-projects.org/.

[7] OpenTracing API. Consistent, expressive, vendor-neutral apis for distributed tracing and context propagation.

[8] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[9] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 15–15, Berkeley, CA, USA, 2003. USENIX Association.

[10] Cristina Basescu, Raphael M Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, and Jumpei Urakawa. Sibra: Scalable internet bandwidth reservation architecture. *arXiv preprint arXiv:1510.02696*, 2015.

[11] Ang Chen, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. Dispersing asymmetric ddos attacks with splitstack. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 197–203, New York, NY, USA, 2016. ACM.

[12] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[13] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 246–256, New York, NY, USA, 2018. ACM.

[14] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 343–359, Berkeley, CA, USA, 2018. USENIX Association.

[15] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Bob DiMaiolo, Jingyu Qian, Chirag Shah, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. Dedos: Defusing dos with dispersion oriented software. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 712–722, New York, NY, USA, 2018. ACM.

[16] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: Early detection of application-level resource exhaustion and starvation attacks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 515–537, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[17] Mohamed Elsabagh, Dan Fleck, Angelos Stavrou, Michael Kaplan, and Thomas Bowen. Practical and accurate runtime application protection against dos attacks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 450–471. Springer, 2017.

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[19] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 817–832, Berkeley, CA, USA, 2015. USENIX Association.

[20] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[22] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 519–530, New York, NY, USA, 2015. ACM.

[23] Gkbrk. SlowLoris attack tool. https://github.com/gkbrk/slowloris.

[24] Dan Goodin. US service provider survives the biggest recorded ddos in history, 2018.

[25] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Comput. Netw.*, 121(C):25–36, July 2017.

[26] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 287–300, Berkeley, CA, USA, 2005. USENIX Association.

[27] Min Suk Kang and Virgil D. Gligor. Routing bottlenecks in the internet: Causes, exploits, and countermeasures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 321–333, New York, NY, USA, 2014. ACM.

[28] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. The crossfire attack. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 127–141, Washington, DC, USA, 2013. IEEE Computer Society.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[30] libuv. A multi-platform support library with a focus on asynchronous i/o.

[31] Xin Liu, Xiaowei Yang, and Yong Xia. Netfence: preventing internet denial of service from inside out. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[32] SS Jeremy Long. Owasp dependency check, 2015. Accessed: 2017-06-11.

[33] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 589–603, Berkeley, CA, USA, 2015. USENIX Association.

[34] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4):11:1–11:28, December 2018.

[35] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[36] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 393–410, Berkeley, CA, USA, 2018. USENIX Association.

[37] Rui Miao, Minlan Yu, and Navendu Jain. Nimbus: Cloud-scale attack detection and mitigation. *SIGCOMM Comput. Commun. Rev.*, 44(4):121–122, August 2014.

[38] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[39] Tongguang Ni, Xiaoqing Gu, Hongyuan Wang, and Yu Li. Real-time detection of application-layer ddos attack using time series analysis. *J. Control Sci. Eng.*, 2013:4:4–4:4, January 2013.

[40] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007.

[41] Open Web Application Security Project. Owasp top ten project'17, 2018. Accessed: 2018-09-27.

[42] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *SIGOPS Oper. Syst. Rev.*, 36(SI):45–60, December 2002.

[43] Steve Ranger. Github hit with the largest ddos attack ever seen, 2018.

[44] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, February 2009.

[45] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[46] David Senecal. Slow DoS on the rise. https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html.

[47] N. Seriot. http://seriot.ch/parsing_json.php, 2016.

[48] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 225–235, New York, NY, USA, 2018. ACM.

[49] 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, Technical report, Google, Inc, 2010.

[50] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages, 2016.

[51] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 361–376, Berkeley, CA, USA, 2018. USENIX Association.

[52] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *Networked and Distributed Systems Security*, NDSS'18, 2018.

[53] Michael Stepankin. [demo.paypal.com] node.js code injection (rce), 2016. Accessed: 2018-10-05.

[54] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 37–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[55] 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. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[56] Vern Paxson Steven McCanne Van Jacobson, Sally Floyd. Tcpdump, a command-line packet analyzer.

[57] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Proceedings of the 25th Networked and Distributed Systems Security Symposium*, NDSS'18, 2018.

[58] Chenxu Wang, Tony TN Miu, Xiapu Luo, and Jinhe Wang. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security*, 13(3):559–573, 2018.

[59] wikipedia. Wikipedia, the free encyclopedia.

[60] Yang Xu and Yong Liu. Ddos attack detection under sdn context. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[61] Xiaowei Yang, David Wetherall, and Thomas Anderson. A dos-limiting network architecture. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 241–252, New York, NY, USA, 2005. ACM.

[62] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

[63] Wei Zhou, Weijia Jia, Sheng Wen, Yang Xiang, and Wanlei Zhou. Detection and defense of application-layer ddos attacks in backbone web traffic. *Future Generation Computer Systems*, 38:36–46, 2014.

SemperOS: A Distributed Capability System

Matthias Hille[†] Nils Asmussen[† *] Pramod Bhatotia[‡] Hermann Härtig[† *]
[†]*Technische Universität Dresden* [‡]*The University of Edinburgh* [*] *Barkhausen Institut*

Abstract

Capabilities provide an efficient and secure mechanism for fine-grained resource management and protection. However, as the modern hardware architectures continue to evolve with large numbers of non-coherent and heterogeneous cores, we focus on the following research question: *can capability systems scale to modern hardware architectures?*

In this work, we present a scalable capability system to drive future systems with many non-coherent heterogeneous cores. More specifically, we have designed a distributed capability system based on a HW/SW co-designed capability system. We analyzed the pitfalls of distributed capability operations running concurrently and built the protocols in accordance with the insights. We have incorporated these distributed capability management protocols in a new microkernel-based OS called SEMPEROS. Our OS operates the system by means of multiple microkernels, which employ distributed capabilities to provide an efficient and secure mechanism for fine-grained access to system resources. In the evaluation we investigated the scalability of our algorithms and run applications (Nginx, LevelDB, SQLite, PostMark, etc.), which are heavily dependent on the OS services of SEMPEROS. The results indicate that there is no inherent scalability limitation for capability systems. Our evaluation shows that we achieve a parallel efficiency of 70% to 78% when examining a system with 576 cores executing 512 application instances while using 11% of the system's cores for OS services.

1 Introduction

Capabilities are unforgeable tokens of authority granting rights to resources in the system. They can be selectively delegated between constrained programs for implementing the principle of least authority [48]. Due to their ability of fine-grained resource management and protection, capabilities appear to be a particularly good fit for future hardware architectures, which envision byte granular memory access to large memories (NVRAM) from a large numbers of cores (e.g. The Machine [31], Enzian [18]). Thereby, capability-based systems have received renewed attention recently to provide an efficient and secure mechanism for resource management in modern hardware architectures [5, 24, 30, 36, 44, 64, 67].

Today the main improvements in compute capacity are achieved by either adding more cores or integrating accelerators into the system. However, the increasing core counts exacerbate the hardware complexity required for global cache coherence. While on-chip cache coherence is likely to remain a feature of future hardware architectures [45], we see characteristics of distributed systems added to the hardware by giving up on global cache coherence across a whole machine [6, 28]. Additionally, various kinds of accelerators are added like the Xeon Phi Processor, the Matrix-2000 accelerator, GPUs, FPGAs, or ASICs, which are used in numerous application fields [4, 21, 32, 34, 43, 60]. These components also contribute to the number of resources an OS has to manage.

In this work, we focus on capability-based systems and how their ability to implement fine-grained access control combines with large systems. In particular, we consider three types of capability systems: L4 [30], CHERI [67] (considered for The Machine [31]), and M^3 [5] (see Section 2.1). For all these capability types it is not clear whether they will scale to modern hardware architectures since the scalability of capability systems has never been studied before. Also existing capability schemes cannot be turned into distributed schemes easily since they either rely on centralized knowledge, cache-coherent architectures, or are missing important features like revocation.

Independent of the choice which capability system to use, scaling these systems calls for two basic mechanisms to be fast. First, it implies a way of concurrently updating access rights to enable fast decentralized resource sharing. This means fast *obtaining* or *delegating* of capabilities, which acquires or hands out access rights to the resources behind the capabilities. The other performance-critical mechanism is the revocation of capabilities. *Revoking* the access rights should be possible within a reasonable amount of time and with minimal overhead. The scalability of this operation is tightly coupled to the enforcement mechanism, e.g. when using L4 capabilities the TLB shootdown can be a scalability bottleneck.

We base our system on a hardware/software co-designed capability system (M^3). More specifically, we propose a scalable distributed capability mechanism by building a multikernel OS based on the M^3 capability system. We present a detailed analysis of possible complications in distributed capability systems caused by concurrent updates. Based on this investigation we describe the algorithms, which we implemented in our prototype OS—the SEMPEROS multikernel.

Our OS divides the system into groups, with each of them being managed by an independent kernel. These independently managed groups resemble islands with locally managed resources and capabilities. Kernels communicate via messages to enable interaction across groups. To quickly find objects across the whole system—a crucial prerequisite for our capability scheme—we introduce an efficient addressing scheme.

Our system design aims at future hardware, which might connect hundreds of processing elements to form a powerful rack-scale system [28]. To be able to experiment with such systems, we use the gem5 simulator [14] Our evaluation focuses on the performance of the kernel, where we showcase the scalability of our algorithms by using microbenchmarks as well as OS-intensive real-world applications. We describe trade-offs in resource distribution between applications and the OS to determine a suitable configuration for a specific application. We found that our OS can operate a system which hosts 512 parallel running application instances with a parallel efficiency of 70% to 78% while dedicating 11% of the system to the OS and its services.

To summarize, our contributions are as follows.

- We propose a HW/SW co-designed distributed capability system to drive future systems. Our capability system extends M^3 capabilities to support a large number of non-cache-coherent heterogeneous cores.
- We implemented a new microkernel-based OS called SEMPEROS that operates the system by employing multiple microkernels and incorporates distributed capability management protocols.
- We evaluated the distributed capability management protocols by implementing the HW/SW co-design for SEMPEROS in the gem5 simulator [14] to run real applications: Nginx, SQLite, PostMark, and LevelDB.

2 Background

We first assess existing capability systems and explain the basic principles of M^3, which is the foundation of our work.

2.1 Capability Systems

The term capability was first used by Dennis and van Horn [20] to describe a pointer to a resource, which provides the owner access to the resource. There are three basic types of capabilities: (1) partitioned capabilities, which have been employed in multiple OSes such as KeyKOS [27], EROS [56] and various L4 microkernels [30,36,40,62], (2) instruction set architecture (ISA) capabilities, as implemented by the CAP computer [49] and recently revived by CHERI [67] and (3) sparse capabilities which are deployed in the password-capability system of the Monash University [2] and in Amoeba [63].

	L4	M^3	CHERI
Scope	Coherence Dom.	Machine	Address space
Enforcement	MMU / Kernel	DTU / Kernel	CHERI co-proc.
Limitation	Coherence Dom.	Core count	no revoke

Table 1: Classification of capability types.

Capabilities can be shared to exchange access rights. ISA capabilities and sparse capabilities can be shared without involving the kernel since their validity is either ensured by hardware or checked by the kernel using a cryptographic one-way function in the moment they are used. In contrast, sharing of partitioned capabilities is observed by the kernel.

To analyze capability systems regarding their scalability, we inspect their enforcement mechanism, their scope, and their limitation in Table 1. The three categories of capability systems in Table 1 represent a relevant subset of capability systems for the scope of this work: (1) L4 capabilities which are partitioned capabilities employed in L4 μ-kernels [30,36,40,62], (2) M^3 capabilities which are a special form of partitioned capabilities involving a different enforcement mechanism explained in the following, and lastly (3) CHERI capabilities which are ISA capabilities implemented by the CHERI processor [64,67].

L4 capabilities utilize the memory management unit (MMU) of the processor to restrict a program's memory access. Access to other resources like communication channels or process control are enforced by the kernel. Since L4 is built for cache coherent machines both the scope of a capability and the current limitation is a coherence domain.

In contrast, M^3 introduces a hardware component called data transfer unit (DTU) which provides message passing between processing elements and a form of remote direct memory access. Consequently, memory access and communication channels are enforced by the DTU and access to other system resources by the kernel. The DTU is the only possibility for a core to interact with other components. Hence it can be used to control a core's accesses to system resources via NoC-level isolation. (We will give a more detailed explanation of M^{3}capabilities in the following section.) Importantly, M^3 capabilities are valid within a whole machine spanning multiple coherence domains. However, M^3 is currently limited by using a single kernel core to control multiple application cores.

Lastly, the ISA capabilities of the CHERI system are enforced by a co-processor. CHERI capabilities contain a description of the resource, i.e., memory they point to. This information is used by the co-processor to determine the validity of accesses. The scope of a CHERI capability is an address space. Thus, such a system typically uses one address space for multiple applications. However, CHERI does not support revocation and therefore does not have the problem we are solving.

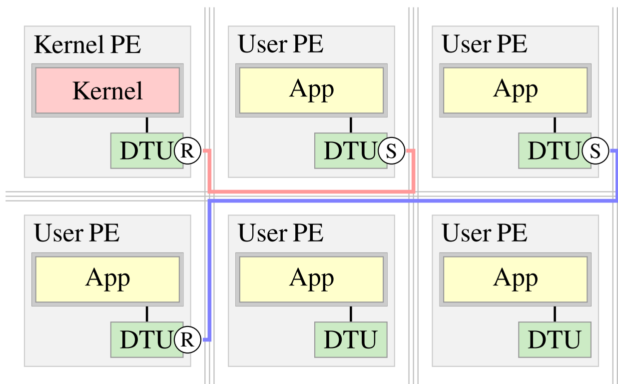


Figure 1: System architecture of M^3. Each processing element (PE) has a data transfer unit (DTU) connecting them to the network-on-chip. DTUs are configured by the kernel PE.

For both L4-style and M^3-style capabilities, scaling to larger systems and maintaining consistency demands the extension to multiple kernels and their coordination. For L4-style systems, multiple kernels are required to scale beyond coherence domains. For M^3-style systems, multiple kernels are required to scale to large core counts.

2.2 M^3: HW/SW Co-designed Capabilities

To accommodate for the hardware trends of growing systems without global cache coherence and an increasingly diverse set of processing elements, we chose M^3 as the foundation of our work. Additionally, M^3 already supports byte-granular memory capabilities including their (selective) revocation (in contrast to CHERI).

The hardware architecture of M^3 is depicted in Figure 1. The key idea of M^3 is to introduce a new hardware component next to each processing element (PE), which is used as an abstraction for the heterogeneity of the PEs, ranging from general purpose cores to accelerators. This hardware component is called data transfer unit (DTU). All PEs are integrated into a network-on-chip (NoC) as prevalent in current multi- and manycore architectures [15, 39, 60]. The DTU represents the gateway for the PE to access remote memory (memories in other PEs or off-chip memory such as DRAM) and to exchange messages with other PEs. As such, the DTU enables a different isolation mechanism, called *NoC-level isolation*, that does not require the PEs to possess hardware features like MMUs and privileged mode execution to ensure isolation. Instead, since all communication between the PEs and all memory accesses are performed via the NoC, controlling the access to the NoC suffices to control and isolate the PEs.

The M^3 kernel runs on a dedicated PE, called *kernel PE*. The M^3 kernel is different from traditional kernels because it does not run user applications on the same PE based on user/kernel mode and entering the kernel via system call, interrupt, or exception. Instead, the kernel runs the applications on other PEs, called *user PEs*, and waits for system calls in the form of messages, sent by the applications via the DTU (red communication channel in Figure 1). Because there is only a single privileged kernel PE in M^3 this kernel PE quickly becomes the limiting factor when scaling to large systems.

Data Transfer Unit (DTU). The DTU provides a number of *endpoints* to connect with other DTUs or memory controllers. Endpoints can represent send, receive or memory endpoints. Establishing communication channels requires to configure endpoints to these representations. This can only be done by a privileged DTU. Initially all DTUs in the system are privileged and get downgraded by the kernel during boot up. Only the DTU of the kernel PE remains privileged. The kernel is required to establish the communication channels between applications (blue in Figure 1) which can be used by applications later on without involving the kernel.

Operating system. The M^3 OS follows a microkernel-based approach, harnessing the features of the DTU to enforce isolation at NoC-level. So far, it employs a single kernel PE to manage the system. M^3 implements drivers and OS services such as filesystems as applications, like other microkernel-based OSes. The execution container in M^3 is called *virtual PE* (VPE), which represents a single activity and is comparable to a single-threaded process. Each VPE has its own capability space and the M^3 kernel offers system calls to create, revoke, and exchange capabilities between the VPEs.

Services on M^3. Services are registered at the M^3 kernel and offer an IPC interface for clients. Additionally, clients can exchange capabilities with services. For example, M^3's in-memory file system, *m3fs*, offers an IPC interface to open files and perform meta operations such as `mkdir` and `unlink`. To access the files' data, the client requests memory capabilities from m3fs for specific parts of the file. The client can instruct the kernel to configure a memory endpoint for the memory capability to perform the actual data access via the DTU, without involving m3fs or the kernel again. This works much like memory mapped I/O, but with byte granular access. Reading the files' data via memory capabilities without involving the OS lends itself well for upcoming non-volatile memory (NVM) architectures.

3 Design

Our design strives to build a scalable distributed capability management for an operating system that uses multiple kernels. An application's scalability depends on two OS components: the kernel itself, especially the capability subsystem, and the OS services, e.g. a filesystem service. To investigate distributed capability management we concentrate on the kernel. The kernel sets up communication channels and memory mappings. How a service implementation uses the kernel mechanisms depends on the type of service. A copy-on-write filesystem for example can be implemented efficiently on top of a capability system with a sufficiently fast revoke operation.

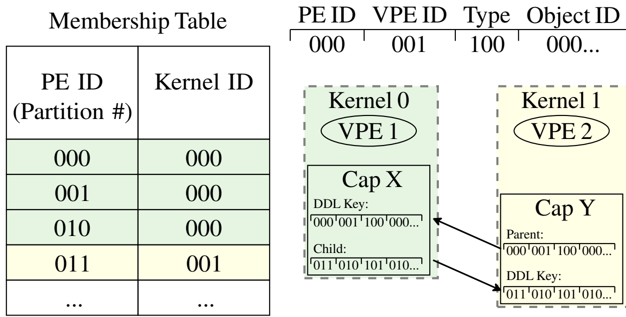


Figure 2: DDL addressing with globally valid DDL keys.

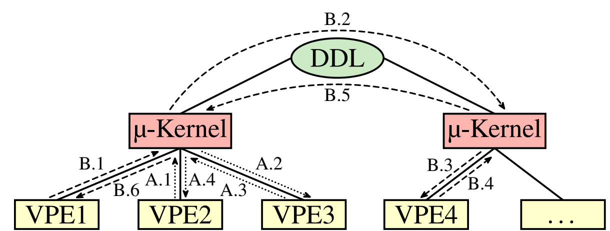


Figure 3: Two VPEs establish a communication channel. Sequence A shows the group-internal communication, whereas sequence B is group-spanning involving two kernels.

When an application performs a write it receives a mapping to its own copy of data and access to the original data has to be revoked. In a capability system with slow revocation it is questionable whether an efficient implementation of a copy-on-write filesystem is possible. The distributed capability system presented in this work shall lay a foundation for various service implementations, however, a discussion on the scaling of OS services is out of scope for this work.

3.1 System Overview

SEMPEROS employs multiple μ-kernels, each kernel running on a dedicated PE. This way we can distribute the handling of system calls to improve the scalability of the capability system. We use message passing as communication means between the kernels since we are not assuming a cache coherent system.

PE groups. To maintain a large number of PEs, we divide them into groups. Each group is managed by a single kernel; that means, every group has to contain at least one PE capable of executing a kernel. A simple general-purpose core is sufficient for this purpose. The group's kernel has exclusive control over all PEs of its group and manages the corresponding capabilities. The mapping of a PE's capabilities to a kernel is static in the current implementation of SEMPEROS because we do not yet support the migration of PEs. The unit of execution being scheduled on a PE is a virtual PE (VPE). A VPE is in general comparable to a process. All system calls of a VPE are handled by the kernel responsible for the PE, which the VPE is running on. If a system call does not affect any VPEs outside the group, only the group's kernel is involved in handling the request. Operations covering the VPEs of other groups involve their kernels as well. A more detailed view on the communication to handle system calls is given in Section 3.3.

Distributed state. The system state consists of the hardware components (available PEs), the PE groups, and the resource allocations and permissions, represented as capabilities. The capabilities represent VPEs, byte-granular memory mappings, or communication channels. Our high-level approach to manage the aforementioned kernel data is to store it where the data emerges and avoid replication as far as possible. Thereby, we minimize the shared state, which reduces the communication required to maintain the coherence.

3.2 Distributed Data Lookup (DDL)

The distributed data lookup is our capability addressing scheme and the mechanism to determine the location of kernel data. Each kernel object or capability which needs to be referable by other kernels is given a *DDL key* which acts as its global ID. In the top right corner of Figure 2 we illustrate how we split the key's value into several regions representing the following: the *PE ID* and *VPE ID*, denoting the creator of the object, and the *Type* and *Object ID*, describing the object itself. To clarify the concept the amount of digits is deliberately kept smaller than our system requires in practice. We use the PE ID to split the key space into multiple partitions. Each PE in the system is allocated to one such partition, which in turn are assigned to kernels individually. The mapping of partitions to the kernels designates the PE groups and is stored in a membership table which is present at each kernel and depicted on the left of Figure 2. To support the migration of PEs, in SEMPEROS, the mappings in the membership table would have to be updated at all kernels of the system However, our current implementation does not support migration yet.

With this addressing scheme we are able to reference objects, for example capabilities, across the whole system which is a key enabler for our capability scheme. The lower right part of Figure 2 illustrates how the relations between the capabilities are tracked by this means. It depicts a situation in which two applications, VPE 1 and VPE 2, are residing in different PE groups, thus managed by different kernels. The (simplified) excerpt of the kernels' internal capability mappings shows that VPE 1 has a capability Cap X and delegated it to VPE 2, which in turn created Cap Y. This group-spanning relation is tracked using the DDL keys.

3.3 System Call Handling

System calls are implemented by messages sent to the kernel PE. Some of the actions taken by the kernel receiving a system call involve agreement between the kernels depending on the involved VPEs. Hence, we divide the handling of system calls into group-internal and group-spanning operations. The different message sequences are outlined in Figure 3.

Group-internal operations. Sequence A in Figure 3 depicts the establishing of a communication channel between two

VPEs (2 & 3) in the same group. Such an operation only involves resources managed by a single kernel, thus it is called a group-internal operation. The sequence starts with a message to the kernel (A.1), which is the counterpart to the traditional mode switch. The connection request is forwarded to VPE3 (A.2) which then responds to the kernel. Depending on the response, the kernel hands out the appropriate capabilities and informs VPE2 (A.4). Once the endpoint for the exchanged communication capability is configured, the kernel is not involved in further communication.

Group-spanning operations. Sequence B in Figure 3 shows the message flow when VPEs of different PE groups establish a communication channel. The second kernel is involved in the operation because it is in charge of VPE4's capabilities. This is where our distributed capability protocol will be used. After receiving the system call in step B.1, the first kernel uses the DDL to determine which kernel is responsible for VPE4 in step B.2 and forwards the request. Steps B.3 and B.4 are identical to A.2 and A.3 of the group-internal operation. After these steps, the channel at VPE4's side is prepared which the first kernel indicates to VPE 1. As for the group-internal operations, the communication via the created channel does not involve the kernel anymore after the endpoint has been configured.

3.4 Capabilities

SemperOS employs capabilities for managing permissions. Each VPE has its own set of capabilities, describing the resources it can access. To share resources with other VPEs, capabilities can be exchanged. In SemperOS delegating a capability starts with a system call by the supplying VPE indicating to the kernel which capability should be delegated. The kernel makes a copy of the selected capability and adds that copy to the capability's children. The copied capability (the child) is handed over to the VPE which shall receive the new access rights. If the sharing is no longer desired, the access rights to the resource can be withdrawn recursively by revoking the capability. These operations require action of the kernel.

From the kernel's perspective, a capability references a kernel object, a VPE, and other capabilities. The kernel object is the resource this capability permits access to and the reference to the VPE (which is a kernel object of its own) tells the kernel who holds these access rights. Individual references to other capabilities are maintained to track sharing as is done by the mapping database in other microkernel-based systems [30, 36, 40, 62]. SemperOS keeps sharing information in a tree structure which is used to enable recursive revocation. In the capability tree, capabilities of different VPEs are interlinked, as indicated in Figure 2.

Challenges in a distributed setting. Capabilities are modified by multiple operations, requested via system calls from applications. For example, when creating a VPE, a capability to control the new VPE is delegated to the parent VPE. We call such actions *capability modifying operations* (CMO). Running a system with multiple independent kernel instances introduces new properties of capability handling:

1. Multiple CMOs can be in flight at the same time.
2. CMOs can involve the modification of capabilities owned by other kernel instances.
3. A capability managed by one kernel can reference a kernel object owned by another kernel.

The first property requires to assure that the modifications of one kernel do not overlap with changes of another kernel. The second and third property are results of our system's distributed nature. Resources such as a service, which is resembled by a service capability, could be used by VPEs of different PE groups. Consider the connection establishment to a service. Assuming that the service is controlled by kernel 1 and the connecting VPE by kernel 2, kernel 2 would create a session capability. A session can only be created between a client and a service; hence, the client's session capability is listed as a child of the service capability. This modification of the service capability's list of children involves the other kernel, because the service capability is owned by kernel 1.

Since capabilities are used to control access to resources, we host a capability at the kernel which owns the resource. Yet, this attribution is not always obvious. The example of a session capability illustrates the third property. One could argue, that the session is a resource which is used by both, the service and the client; thus, any of the two corresponding kernels could actually be responsible for the session. To avoid the overhead of coordination between multiple resource owners, we allow only one kernel to be the owner of a resource.

4 Implementation

SemperOS implements a distributed capability scheme with multiple kernels in order to scale to large numbers of PEs. We based SemperOS on M^3 [5]. SemperOS adds PE groups to the base system, requiring coordination of the kernels which we implement by so called inter-kernel calls explained in the following Section 4.1. Furthermore, SemperOS is implemented as a multithreaded kernel for reasons explained in Section 4.2. The transparent integration of the PE groups into a single system presented to the applications requires the kernels to implement a distributed capability system described in Section 4.3.

4.1 Inter-Kernel Calls

The system call interface of SemperOS did not change compared to M^3 though the action to be taken by the kernel changed to incorporate the coordination with other kernels. The kernels in SemperOS communicate via messages adhering to a messaging protocol. We call this type of remote procedure calls *inter-kernel calls*. These calls can be split into three functional groups: (1) messages to start up and shutdown kernels and OS services, (2) messages to create connections to the services in other PE groups, and lastly, (3) messages

used to exchange and revoke capabilities across PE-group boundaries. Messages of the last two groups are part of the distributed capability protocol.

The DTU, which is used to send and receive messages, provides only a limited number of message slots. If this limit is exceeded then the messages will be lost. To prevent this, we limit the number of in-flight messages between two kernels. We dedicate a certain number of DTU endpoints for the kernel-to-kernel communication, which also determines the maximum number of kernels supported by the system. We keep track of free message slots at each kernel to avoid the message loss.

4.2 Multithreaded Kernel

The kernel needs to split some operations, e.g. revocation or service requests across the PE groups, into multiple parts to prevent deadlocks. For instance, a revocation might run into a deadlock in the following situation: three capabilities are involved forming the capability tree: $A_1 \rightarrow B_2 \rightarrow C_1$. The index indicates the kernel which owns the capability. If A_1 is revoked, kernel 1 contacts kernel 2 which in turn contacts kernel 1 again to revoke capability C_1. If kernel 1 would block on the inter-kernel call to kernel 2, the system would end up in a deadlock because kernel 2 is waiting for kernel 1 to respond. While this can be implemented as an event-driven system, this involves the danger to loose the overview of the logical flow of complicated operations like the revocation.

Therefore, we decided to use cooperative multithreading within the kernel. This approach allowed us to implement such capability operations sequentially with dedicated preemption points in between, which made it comparatively easy to reason about the code. Note that, in contrast to simultaneous multithreading, SEMPEROS only executes one thread per kernel at a time because it executes on one single-threaded core. The preemption points do not only prevent deadlocks, but also allow to process other system calls or requests from other kernels until the suspended operation can be continued.

To prevent the denial-of-service attacks on the kernel, the kernel cannot spawn new threads on behalf of system calls. Instead, a fixed number of threads needs to suffice. We create a kernel's thread pool at start up. The size of the pool is determined by the number of system calls and kernel requests which can arrive at the same time. It is calculated as:

$$V_{group} + K_{max} * M_{inflight} \quad (1)$$

Since each VPE can issue only one system call at a time, the kernel needs one thread per VPE in its PE group, denoted as V_{group}. The number of kernel requests is limited by the maximum amount of kernels in the system, denoted as K_{max}, multiplied by the maximum number of in-flight messages between two kernels, denoted as $M_{inflight}$.

4.3 Distributed Capability Management

SEMPEROS uses capabilities to control the access to resources such as VPEs, service connections, send/receive endpoints and memory. Applications can *create*, *use*, *exchange*, and *revoke* capabilities. Creation means to create a new capability for a given resource (e.g., memory) and usage means to use a previously created capability without changing it (e.g., configure a DTU endpoint for a send capability). Exchanges and revokes are *capability modifying operations* (CMO), requiring the most attention. Exchanging capabilities allows two VPEs to share resources and it comes in two flavors: a capability can be *delegated* to another VPE, and *obtained* from another VPE. Capability exchanges can be undone with the revoke operation. Revocation is performed recursively, that is, if VPE V_1 has delegated a capability to VPE V_2, which in turn has delegated it to V_3 and V_4, and V_1 revokes its capability, it is revoked from all four VPEs.

As in other capability systems [30, 36, 40, 62], the recursive revoke requires a way to track former exchanges. The kernel uses a so-called *mapping database* for this purpose. In SEMPEROS each capability has a parent and a list of children to explicitly track all such links. These tree relations can span multiple kernels; hence, we use DDL keys to identify and locate the capabilities across all kernels. The mapping database is updated on every CMO. In a multikernel setting multiple CMOs can be started concurrently potentially involving the same capability. We next describe how inconsistent updates on the mapping database are prevented if multiple CMOs run in parallel.

4.3.1 Interference between CMOs

Exchanging a capability consists of two actions: (1) creating a new capability based on the donor's capability and (2) inserting the new capability into the capability tree. The latter requires to store a reference to the parent capability and to update the parent's list of children. Revoking a capability requires to revoke all of its children and to remove it from the parent's list of children. Both need to perform inter-kernel calls in case capabilities reside at other kernels, possibly leading to interference.

An important precondition for all operations is that messages between two kernels need to sustain ordering. More specifically, if kernel K_1 first sends a message M_1 to kernel K_2, followed by a message M_2 to kernel K_2, then K_2 has to receive M_1 before M_2.

Table 2 shows an overview of all combinations and their effects. The operation in the leftmost column is started first and overlaps with the operation in the topmost row. The following walks through the combinations and describes the effects.

Serialized. Overlapping exchange operations do not present a problem for our scheme because they serialize at one kernel. For example, if two VPEs obtain a capability from VPE V_1, these operations serialize at the kernel that manages V_1. In general, each VPE can only perform one system call at a time preventing two parallel delegates initiated by the same VPE.

1st \ 2nd	Obtain	Delegate	Revoke/Crash
Obtain	✓Serialized	✓Serialized	! Orphaned
Delegate	✓Serialized	✓Serialized	ϟ Invalid
Revoke	! Pointless	! Pointless	ϟ Incomplete

Table 2: Types of interference with overlapping CMOs.

However, during an exchange operation initiated by a VPE V_1 other VPEs could exchange capabilities with V_1, which again serializes at the kernel that manages V_1.

Orphaned. The obtain operation needs to ask the capability owner for permission before being able to obtain the capability. If the owner resides at a different kernel this requires an inter-kernel call. Before its completion nothing is changed in the obtainer's capability tree. However, the obtainer could be killed while waiting for the inter-kernel call. This leaves an orphaned child capability in the owner's capability tree, in case the owner agreed to the exchange. The orphaned capability cannot be accessed by anyone, but it wastes a bit of memory, which will be freed the latest when its parent capability is revoked.

Invalid. The delegate operation is similar to obtain regarding leaving the delegator's capability tree untouched until the inter-kernel call returns successfully. However, if the delegator is killed while waiting for the inter-kernel call, the receiving VPE might have already received the capability. This constitutes a problem, because the child of the capability in delegation does not yet exist in the delegator's capability tree. That is, although all capabilities of the delegator are revoked, the delegated capability stays valid at the receiving VPE.

Incomplete. The naive implementation of the revoke operation would simply perform a depth-first walk through the capability tree, remove local capabilities on its own and perform the inter-kernel calls to remove remote capabilities. However, if two revoke operations run in parallel on overlapping capability subtrees, this approach results in early replies to revoke system calls, that is, acknowledgements of incomplete revokes.

For instance, let us consider the following capability tree with the owning kernel as the index: $A_1 \rightarrow B_2 \rightarrow C_3$. If a VPE requests the revoke of A_1, kernel K_1 performs a call to K_2 to revoke the remaining part of the tree. If another VPE requested the revoke of B_2 in the meantime, K_2 does not know B_2 anymore, potentially leading to an early response to K_1. The reason is that K_2 might still be waiting for K_3 to revoke C_3. Since applications have to rely on the semantic that completed revokes are indeed completed, we consider this behavior unacceptable.

Pointless. The revoke operation requires inter-kernel calls if the capability tree spans multiple kernels. Hence, VPEs might request capability exchanges of not yet revoked capabilities within this tree. This does not lead to inconsistencies because these capabilities would be revoked as soon as the running revoke operation continues. However, the exchange is pointless because it is already known that the capabilities will be revoked afterwards.

4.3.2 Capability Exchange

This section details the capability exchange operations to address the problems described in the previous section. As already mentioned, the beginning of obtain and delegate is similar. If a VPE (V_1) requests an exchange, the corresponding kernel (K_1) checks whether the other party (V_2) is in the same PE group. If so, the operation is handled by K_1. If V_1 and V_2 are in different PE groups, K_1 forwards the exchange request to the second kernel (K_2). K_2 asks V_2 whether it accepts the capability exchange. If V_2 denies the exchange, the operation is aborted and a corresponding reply is sent to K_1. Otherwise, we distinguish between obtain and delegate:

(1) Obtain:. V_2's capability (C_2) will become the parent of V_1's new capability (C_1). Hence, C_1 will be added to C_2's list of child capabilities. Afterwards, K_2 sends a reply to K_1. As outlined previously, if V_1 was killed in the meantime, C_2 stays in the child capability list as an orphaned capability. To prevent a permanent memory waste, we let K_1 send a notification to K_2 on behalf of K_2's reply for the obtain operation in case V_1 was killed.

(2) Delegate: . K_2 creates a new capability (C_2) for V_2 with C_1 as its parent. If C_1 was revoked in the meantime, V_2's resource access through C_2 would be unjustified. To avoid this, we implement delegation with a two-way handshake. Instead of inserting C_2 into V_2's capability tree, K_2 only sends a reply to K_1. After that K_1 adds C_2 to C_1's list of children and sends an acknowledgement back to K_2 to actually insert C_2 into V_2's capability tree.

Note that the two-way handshake creates an orphaned capability if V_2 is killed while waiting for the acknowledgement of K_1. As for obtain, we handle this case by sending an error back to K_1 to allow a quick removal of the orphaned capability.

4.3.3 Capability Revocation

Like the capability exchange, the revocation requires inter-kernel calls in case the capability tree spans multiple kernels. To keep the kernel responsive it should not wait synchronously for the reply of another kernel. Instead, the revoke should be paused using the threading infrastructure introduced in Section 4.2. However, in contrast to the exchange operation, the number of inter-kernel calls for a revoke can be influenced by applications. For example, two malicious applications residing in different PE groups could exchange a capability back and forth, building a deep hierarchy of capabilities at alternating kernels. Revoking this capability hierarchy would lead to inter-kernel calls sent back and forth between the two kernels. Thus, the naive approach of spawning a new thread for every incoming revoke inter-kernel call cannot be used, because this would enable denial-of-service attacks. Our solution uses a maximum of two threads per kernel to avoid this attack.

To avoid acknowledgements of incomplete revokes, our algorithm uses two phases, similar to mark-and-sweep [46]. Algorithm 1 presents a high-level overview of the approach. The function `revoke_syscall_hdlr` is executed by the

kernel that receives the revoke system call. First, it calls `revoke_children`, which recursively marks all local capabilities and sends inter-kernel calls for remote capabilities. Each capability maintains a counter for outstanding kernel replies. If it encountered any remote capabilities, the function `wait_for_remote_children` waits for the kernel replies by pausing the thread.

The inter-kernel call is handled by `receive_revoke_request`, which will also call `revoke_children`. In this case, the thread will not be paused to stay at a fixed number of threads. Instead, the thread calls `receive_revoke_reply` in case there are no outstanding kernel replies and returns. The function `receive_revoke_reply` is also called whenever a reply to an inter-kernel call is received, which first updates the counter of the capability accordingly. If there are no further outstanding kernel replies, it deletes the capability tree starting at the given capability. Afterwards, it wakes up the syscall thread or sends a reply, depending on whether this kernel started the revoke operation or participated due to an inter-kernel call.

To keep the pseudo code brief it does not show how already running revocations for a capability are handled. In this case, `revoke_syscall_hdlr` will also wait for the already outstanding kernel replies to prevent acknowledgement of an incomplete revoke. Furthermore, the two phases allow us to immediately deny exchanges of capabilities that are in revocation, which prevents pointless capability exchanges.

Algorithm 1: Capability revocation

```
1  Function revoke_syscall_hdlr(capability)
2  |   revoke_children(capability)
3  |   wait_for_remote_children()
4  Function revoke_children(capability)
5  |   mark_for_revocation(capability)
6  |   foreach child of capability do
7  |   |   if child is local then
8  |   |   |   revoke_children(child)
9  |   |   else
10 |   |   |   send_revoke_request(child)
11 |   |   end
12 |   end
13 Function receive_revoke_request(capability)
14 |   revoke_children(capability)
15 |   receive_revoke_reply(capability) // see line 10
16 Function receive_revoke_reply(capability)
17 |   if all revoke requests are serviced then
18 |   |   delete_tree(capability)
19 |   |   if initiator then
20 |   |   |   notify_syscall_hdlr_thread()
21 |   |   else
22 |   |   |   send_revoke_reply()
23 |   |   end
24 |   end
```

Operation	Scope	SemperOS (cycles)	M^3 (cycles)	Increase
Exchange	Local	3597	3250	10.7%
Exchange	Spanning	6484	—	—
Revoke	Local	1997	1423	40.3%
Revoke	Spanning	3876	—	—

Table 3: Runtimes of capability operations.

5 Evaluation

5.1 Experimental Testbed

We evaluate SemperOS using the gem5 system simulator [14], which enables us to evaluate the hardware/software co-designed capability system and perform experiments on systems larger than currently available. The system is composed of 640 out-of-order x86_64 cores, which are clocked at 2 GHz. However, the cores used for applications could also be exchanged with any other architecture or accelerator. Each core is equipped with a DTU similar to the one used in Asmussen et al.'s work [5]. We modified the mechanism to store incoming messages to support delayed replying which enables us to interrupt kernel threads. Messages are kept in a fixed number of slots. Each DTU provides 16 endpoints with 32 message slots each. Kernel PEs use one endpoint to send messages to other kernels, one endpoint to send messages to services, and 14 endpoints to receive messages. Six of the receiving endpoints are used for the system calls. Each kernel can handle up to 192 PEs in the current implementation since each VPE can only issue one (blocking) system call at a time. Eight endpoints are used to receive messages from other kernels. In contrast to system calls, the inter-kernel calls are non-blocking and we limit the number of in-flight messages to four messages per kernel. Thus, at most 64 kernel PEs are supported.

5.2 Microbenchmarks

Capability exchange and revocation. To examine the exchange and revocation of capabilities we start two applications where the second application obtains a capability from the first, followed by a revoke by the first application. We distinguish two scopes for these operations: group-local and group-spanning. In the group-local case one kernel manages both applications and their capabilities. The group-spanning case involves two kernels, each handling one application.

Table 3 lists the execution times in cycles for exchanging and revoking capabilities in the group-local and group-spanning case. We can only compare the group-local case to M^3, because in M^3 there is only one kernel. To support multiple kernels, SemperOS references parent and child capabilities via DDL keys instead of plain pointers. Analyzing the DDL key to determine the capability's owning kernel and VPE introduces overhead in the local case. Group-spanning operations involve another kernel, which almost doubles the time of exchanges and revokes. This suggests, that applications should be assigned to PE groups such that the group-spanning operations are minimized.

Chain revocation. In the chain revocation benchmark we measure the time to revoke a number of capabilities forming a chain. Such a chain emerges when a capability is exchanged with an application which in turn exchanges this capability

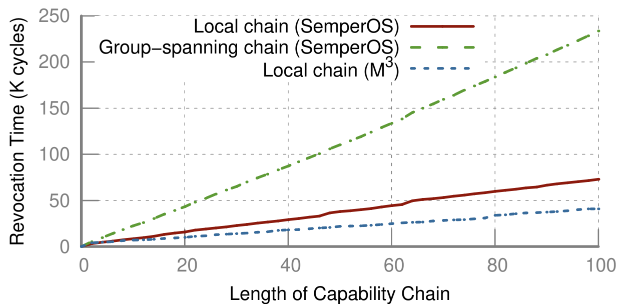


Figure 4: Revoking capability chains of varying sizes.

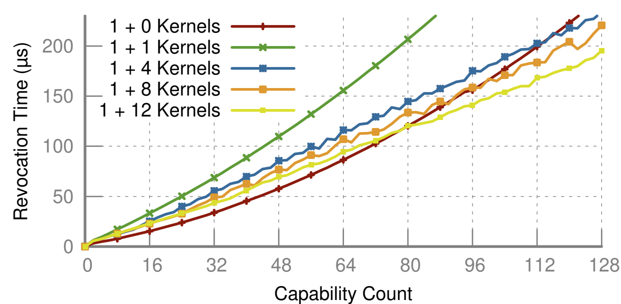


Figure 5: Parallel revocation of capability trees with different breadths utilizing multiple kernels.

again with another application and so on. Figure 4 depicts the time to revoke such capability chains depending on their length. A local chain comprises only applications managed by one kernel and can again be compared to M^3. As the previous microbenchmarks showed, revocation in SEMPEROS needs about twice the time compared to M^3 due to the added indirections.

The group-spanning chain depicts a scenario in which an ill-behaving application repeatedly exchanges a capability between two VPEs, which are managed by different kernels. This creates a circular dependency between the two involved kernels during revocation. However, as described in Section 4.3.3, this is not a problem for our revocation algorithm. In particular, only the kernel thread for the revoke system call is blocked during the operation. Still, the revocation of a group-spanning chain takes about three times longer than revoking a group-local chain, because messages are sent back and forth between the two kernels.

Tree revocation. This microbenchmark resembles a situation, in which an application exchanges a capability with many other applications, for example, to establish shared memory. This results in a capability tree of one root capability with several children. Figure 5 shows the performance of the revocation depending on the capability count and their distribution among kernels. The line labeled with 1 + 0 Kernels represents the local scenario in which the whole capability tree is managed by one kernel. For all other lines, the second number indicates the number of kernels the child capabilities have been distributed to. After exchanging the capabilities, the application owning the root capability revokes the capability tree. Figure 5 illustrates that the revocation scales to many capabilities and kernels. It also shows that our current implementation can take advantage of multiple kernels by performing the revoke in parallel, but the effect is rather small. It currently leads to a break-even at 80 child capabilities, when comparing the local revocation time with a parallel revocation with 12 kernels. However, we believe that this can be further improved by the use of message batching. So far, the kernel managing the root capability sends out one message for each child capability.

5.3 Application-level Benchmarks

We next perform application-level benchmarks to examine the scalability of SEMPEROS in more realistic settings.

5.3.1 Experimental Setup

Applications. We use seven different applications to analyze the scalability: *tar* and *untar* pack or unpack an archive of 4 MiB containing five files of sizes between 128 and 2048 KiB. The *find* benchmark scans a directory tree with 80 entries for a non-existent file. The *SQLite* database and the *LevelDB* key-value store both create a table to insert 8 entries into it and select them afterwards. The PostMark mailserver application resembles a heavily loaded mail server, thus does a lot of operations on the mail files. (In addition, we evaluated *Nginx* Webserver in Section 5.3.3.) Note that we were forced to use rather short running applications to keep the simulation times of gem5 acceptable (e.g. SQLite required five days on a 48-core machine). The selected applications are well suited for this evaluation since they make heavy use of the OS in various ways. In particular they use the in-memory filesystem service which implements file access by handing out the memory capabilities to the clients so they can access the memory region in which the requested file is stored. More specifically, the filesystem service hands out a memory capability to a range of the file's contents. If the application exceeds this range, for example by appending to the file, it is provided with an additional memory capability to the next range. When the file is closed again, the memory capabilities are revoked.

Table 4 lists the number of capability operations for the individual benchmark applications. We show the numbers for a single benchmark instance and 512 parallel instances. The capability operations per second for 512 benchmark instances are retrieved when employing 64 kernels and 64 filesystem services; we will explain what this means in the following paragraph on our methodology. The tar and untar benchmarks are memory-bound applications exposing a regular read and write pattern which requires the filesystem service to hand out several memory capabilities throughout the benchmark execution. The find benchmark mainly stresses the filesystem service by doing many stat calls to examine the directory's metadata. The small database engine of SQLite exhibits a more compute intensive behavior with several bursts of capability operations when opening and closing the database and the database journal whereas the LevelDB key-value store accesses its data files with a higher frequency resulting

Benchmark	Cap. ops	Cap. ops/s	Cap. ops	Cap. ops/s
# of instances	1		512	
tar	21	7,295	10,752	191,703
untar	11	4,012	5,632	100,772
find	3	1,310	1,536	27,096
SQLite	24	5,987	12,288	207,072
LevelDB	22	8,749	11,264	201,204
PostMark	38	21,166	19,456	348,285

Table 4: Number of capability operations for the selected applications. Values shown for 1 and 512 parallel benchmark instances. The capability operations per second are the average rate of capability operations over the runtime.

in more capability operations per second. PostMark does little computation and operates on many files resulting in the highest load for the capability system.

Performance metric. We use the system call tracing infrastructure introduced by Asmussen et al. [5] to run the benchmarks. We run an application on Linux, trace the system calls including timing information, and replay the trace on SEMPEROS while checking for correct execution. We account for the system calls which are not supported by our system yet by waiting for the time it took to execute them on Linux. However, all relevant system calls (especially those to interact with the file system) are executed. We replay the same trace multiple times in parallel, which is denoted as number of benchmark or application instances in the graphs. We assess scalability using the parallel efficiency of these benchmark instances. In a perfectly scaling system, a benchmark instance will have the same execution time when running alone as when running with other instances in parallel. However, due to resource contention for the kernel and for hardware resources like the interconnect and the memory controller, each instance will need more time if multiple of them are executed in parallel. The discrepancy in runtime is shown by *parallel efficiency*.

Methodology. There are two main factors influencing the scalability of applications running in a μ-kernel-based system: the OS services and the kernel. The OS service used by the examined applications is the m3fs filesystem. To concentrate our analysis on the scalability of the kernel, especially the distributed capability management, we simplify scaling of the m3fs service by adding more service instances, each having its own copy of the filesystem image in memory. We exclude accessing the actual memory locations of the files because our current simulator does not include a scalable memory architecture yet. Instead we let the application compute for the amount of time the access would have taken, assuming a non-contended memory controller. We argue that this still produces useful results since we do not want to show the scalability of the memory architecture but of the distributed capability scheme. Furthermore, a non-contended memory puts even more burden on the OS because capability operations might occur with higher frequency.

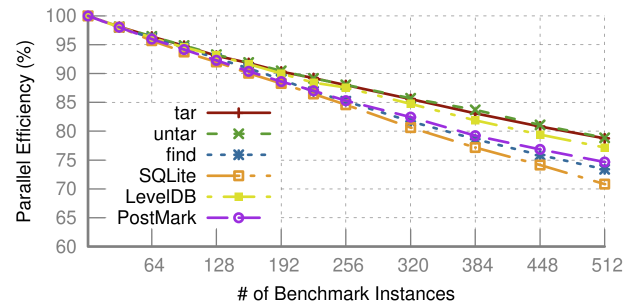


Figure 6: Parallel efficiency of all six applications using 32 kernels and 32 file service instances.

5.3.2 Results

Scalability. Figure 6 depicts the parallel efficiency of the six applications when distributing them equally between 32 kernels and 32 filesystem services. With this configuration the tar benchmark already reaches an efficiency of 78% when running 512 instances in parallel. However, SQLite achieves only 70%, which is not the optimal configuration for this type of application as we will show in the next measurement (see Figure 7). We next discuss how to determine a fitting configuration for an application.

Service dependence. To determine the number of services required to scale an application we set the number of kernels to a high number and then gradually increase the number of services. As long as there are less services than kernels, services are shared between PE groups. Kernels which host a service in their PE group prefer to connect their applications to the service in their PE group over a service in another PE group. Figure 7 shows the parallel efficiency for tar and SQLite depending on the number of services.

The tar benchmark is not very dependent on the filesystem service, which can be inferred from the fact that using 48 services does not pose any improvement over 32 services. In fact, it seems already fair enough to use only 16 service instances. SQLite shows a higher dependence on the number of services. For example, increasing the number of service instances from 16 to 32 leads to further improvement of 9 percent points.

Kernel dependence. Similarly to the dependence on the number of services, we now show the influence of the number of kernels. Figure 8 depicts the parallel efficiency of PostMark and LevelDB using a fixed number of services. LevelDB exhibits smaller improvements when employing more than 16 kernels compared to PostMark, indicating that PostMark is even more susceptible to the number of kernels. However, all applications show a relatively high sensitivity to the number of kernels, which in fact are mostly handling capability operations. This confirms our expectation that a scalable distributed capability system is a vital part of a fast μ-kernel-based OS for the future hardware architectures. The analysis so far only involved tuning for parallel efficiency, which is analogous to optimize for execution time. We next discuss the efficient usage of PEs.

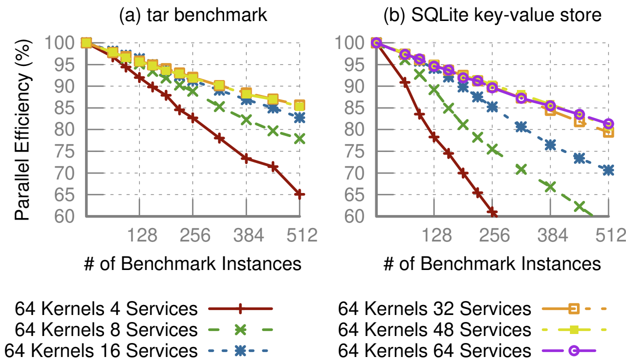


Figure 7: Service dependence: Parallel efficiency of tar and SQLite with fixed number of kernels.

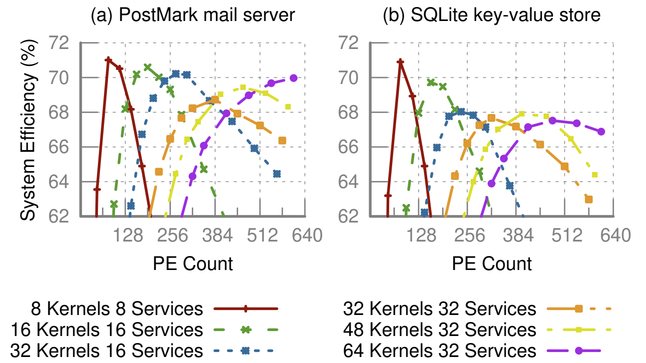


Figure 9: System efficiency of PostMark and SQLite with different configurations.

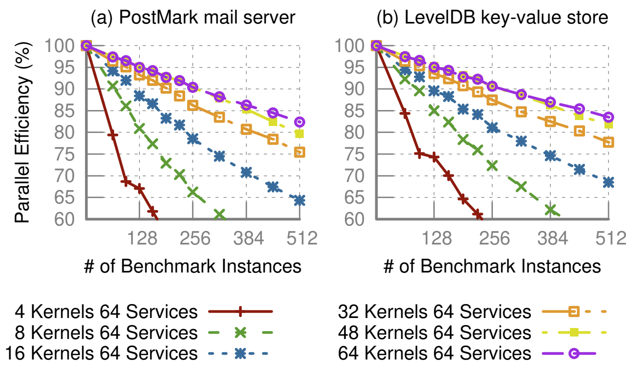


Figure 8: Kernel dependence: Parallel efficiency of PostMark and LevelDB with fixed number of services.

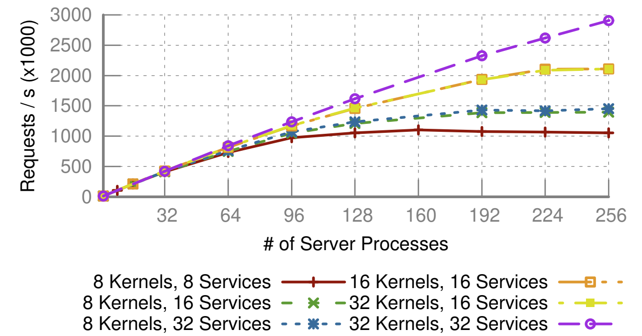


Figure 10: Scalability of the Nginx webserver.

System efficiency. If we consider the whole system and account for the PEs used by the OS with an efficiency of zero, the optimal configurations change. We call this measure the *system efficiency*, which is depicted in Figure 9. Instead of showing the efficiency only in relation to the benchmark instances executed we relate them to the total number of PEs. By means of this metric we can tune a system for throughput and determine the optimal number of kernels and services for an application depending on the number of PEs available. For SQLite this implies to choose 16 kernels and 16 service instances if the system had 192 PEs, but if the system would consist of 256 PEs we would run it with 32 kernels and 16 services.

5.3.3 Server Benchmark

We next detail the results for the Nginx webserver [53]. We used our system call tracing infrastructure to record the behavior of Nginx on Linux when handling requests. We stressed Nginx similar to the Apache ab benchmark [1] by introducing PEs that resemble a network interface. These PEs constantly send out requests to our webserver processes running on separate PEs. These PEs replay the trace upon receiving a request and send the response back. Figure 10 depicts the number of requests per second of all webserver PEs. Despite this OS-intensive benchmark, the number of requests scales almost linearly when employing 32 kernels and 32 services. Using less resources for the OS flattens the graph.

6 Related Work

Capability systems. The evaluation of previous capability systems typically resorted to performance measurements of single core or small systems. Since many capability systems, such as Mach, Fluke or EROS [19,23,56] are based on μ-kernels which had to prove their enhanced efficiency over previous μ-kernel generations [29,41], their kernel mechanisms like inter-process communication, system-call performance, context-switch overhead or process-creation time have been measured to demonstrate their in fact competitive performance. With these measurements it is only partly possible to derive the performance of the capability subsystem. So far the only work which included capabilities in a distributed setting is Barrelfish [11] but only parts of the capability subsystem's scalability can be concluded from the reported results. Barrelfish's two-phase commit approach to reach agreement (determining the relations between capabilities) requires broadcasting to every other kernel in the system, which is different from our approach. The revocation in Barrelfish uses a mark-sweep algorithm and

so called *delete cascades* which also require to broadcast to every kernel if capabilities have cross-kernel relations because these are not stored explicitly in Barrelfish [25]. Even though this broadcast operation can be tuned to fit the interconnect of the machine it is running on [33], it is unknown how well it performs in conjunction with their capability scheme.

Other capability systems like Capsicum [65] and CHERI [67] emphasize their sandboxing features and compatibility to existing software by executing application benchmarks. However, these do not include any assessment of large scalable systems. Further, CHERI does not support revocation, thus eliminating the overhead of tracking capability relations.

Operating systems. Apart from a scalable capability subsystem the OS also has to entail mechanisms to drive large possibly heterogeneous systems. The monolithic architecture of Linux, which runs a shared-memory kernel on homogeneous cores, has many scalability bottlenecks [16, 17, 26]. Developers try to counteract that by utilizing scalable data structures like RCU [47] within the kernel. To investigate more profound changes researchers built frameworks like K42 [37] to enable development and testing of new approaches like clustered objects [3]. Song et al. proposed Cerberus [61] which runs multiple Linux instances on top of a virtualization layer.

Systems like Popcorn Linux [8–10], and K2 [42] adapt Linux for heterogeneous ISAs. These systems also run multiple Linux instances closely resembling a distributed system. Rack-scale operating systems like LegoOS employ multiple distributed components to manage disaggregated resources [55]. They can benefit from our approach when combined with capabilities.

Barrelfish proposed the multikernel approach which aims to improve the scalability and support for many heterogeneous cores by constructing the OS as a distributed system. Cosh [12], a derivative of Barrelfish, demonstrated how to share and provide the OS services across different domains. While Cosh defines an interface how to communicate between different coherence islands and adds guarantees regarding memory accesses after sharing, it is not discussing the underlying capability system. Barrelfish/DC [68] examined the separation of kernel state from a kernel instantiation to provide an elastic system. Fos [66] targets manycore systems by proposing the concept of service fleets to provide OS services via spatially distributed servers. Importantly, these OSes are based on communication over message passing and do not assume cache coherent shared memory. Our system SEMPEROS shares the same two design principles: (1) the multikernel approach, and (2) communication via message passing. However, the design of Barrelfish and fos require executing a kernel on every core. Whereas, we based our work on M^3 [5] which runs the kernel only on a single dedicated core. This allows us to explore another design point in the multikernel design space in which the capabilities of several processing units are managed by one kernel which has to coordinate with other kernels to scale to large systems.

The philosophy of providing OS services without executing a kernel on every core has also been explored in NIX [7], which is based on Plan 9 [52]. NIX proposes an OS with a concept of application cores which do not execute a kernel to prevent OS noise. However, the communication in NIX is based on shared memory. Similarly, Helios [50], an extension of Singularity [22], minimizes the kernel requirement for some cores to a smaller satellite kernel.

Motivated by the recent trends in hardware, in a similar spirit but with a different focus, new OSes such as Arrakis [51], IX [13], and Omnix [57] have been proposed. These OSes share a similar design philosophy to SEMPEROS where we aim to provide applications direct control of the underlying hardware to improve the performance.

Alternatively, there are several proposals to support OS services for one specific type of accelerator. For instance, GPUfs [58], GPUNet [35], and PTask [54] are designed for GPUs. Likewise, BORPH [59], FPGAFS [38], etc. are designed to support FPGAs. In contrast, using M^3 as our foundation allows us to support different types of accelerators and general purpose heterogeneous cores as first-class citizens.

7 Conclusion

In this paper, we presented a HW/SW co-designed distributed capability system based on M^3. More specifically, we presented a detailed analysis of distributed capability management, covering the inconsistencies which can arise in a distributed multikernel setting where concurrent updates to capabilities are possible. Leveraging the results of this investigation we devised efficient algorithms to modify capabilities in a scalable and parallel manner.

We implemented these algorithms in our microkernel-based OS, SEMPEROS, which employs multiple kernels to distribute the workload of managing the system. We evaluated the distributed capability management protocols by co-designing the HW/SW capability system in the gem5 simulator [14]. Our evaluation shows that there is no inherent scalability limitation for capability systems for running real applications: Nginx, SQLite, PostMark, and LevelDB. In particular, we showed that SEMPEROS achieves a parallel efficiency of 70% to 78% when running 512 applications and dedicating 11% of the system's cores to the OS.

Software availability. SEMPEROS will be open-sourced at https://github.com/TUD-OS/SemperOS.

8 Acknowledgements

We would like to thank our shepherd, Gernot Heiser, and the anonymous reviewers for their helpful suggestions. This work was funded through the German Research Council DFG through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed), and by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK, and by public funding of the state of Saxony/Germany.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: May, 2018.

[2] ANDERSON, M., POSE, R., AND WALLACE, C. S. A password-capability system. *The Computer Journal* (1986).

[3] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS)* (2007).

[4] ARNOLD, O., MATUS, E., NOETHEN, B., WINTER, M., LIMBERG, T., AND FETTWEIS, G. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)* (2014).

[5] ASMUSSEN, N., VÖLP, M., NÖTHEN, B., HÄRTIG, H., AND FETTWEIS, G. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[6] BALKIND, J., LIANG, X., MATL, M., WENTZLAFF, D., MCKEOWN, M., FU, Y., NGUYEN, T., ZHOU, Y., LAVROV, A., SHAHRAD, M., FUCHS, A., AND PAYNE, S. OpenPiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[7] BALLESTEROS FRANCISCO J., EVANS NOAH, F. C., AND GUARDIOLA GORKA, MCKIE JIM, MINNICH RON, S.-S. E. NIX: A case for a manycore system for cloud computing. *Bell Labs Technical Journal* (2012).

[8] BARBALACE, A., ILIOPOULOS, A., RAUCHFUSS, H., AND BRASCHE, G. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[9] BARBALACE, A., LYERLY, R., JELESNIANSKI, C., CARNO, A., CHUANG, H.-R., LEGOUT, V., AND RAVINDRAN, B. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[10] BARBALACE, A., RAVINDRAN, B., AND KATZ, D. Popcorn: a replicated-kernel OS based on Linux. *Ottawa Linux Symposium (OLS)* (2014).

[11] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009).

[12] BAUMANN, A., HAWBLITZEL, C., KOURTIS, K., HARRIS, T., AND ROSCOE, T. Cosh: Clear OS data sharing in an incoherent world. In *2014 Conference on Timely Results in Operating Systems (TRIOS)* (2014).

[13] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[14] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *SIGARCH Computer Architecture News* (2011).

[15] BOHNENSTIEHL, B., STILLMAKER, A., PIMENTEL, J., ANDREAS, T., BIN LIU, TRAN, A., ADEAGBO, E., AND BAAS, B. A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *IEEE Symposium on VLSI Circuits (VLSI-Circuits)* (2016).

[16] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)* (2010).

[17] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM TOCS* (2015).

[18] COCK ET AL. Enzian: a research computer for datacenter and rackscale computing. In *Poster proceedings of the 13th European Conference on Computer Systems (EuroSys)* (2018).

[19] DAVID GOLUB, R. D., GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. Unix as an application program. In *In USENIX 1990 Summer Conference* (1990), pp. 87–95.

[20] DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM* (1966).

[21] DONGARRA, J. Report on the Tianhe-2A system. Tech. rep., University of Tennesssee Oak Ridge National Laboratory, 2017.

[22] FÄHNDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J. R., AND LEVI, S. Language support for fast and reliable message-based communication in singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (2006).

[23] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)* (1996).

[24] GE, Q., YAROM, Y., CHOTHIA, T., AND HEISER, G. Time protection: the missing OS abstraction. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (2019).

[25] GERBER, S. *Authorization, Protection, and Allocation of Memory in a Large System*. PhD thesis, ETH Zurich, 2018.

[26] HAIBO, S. B.-W., RONG, C., YANDONG, C., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., AND WU, M. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2008).

[27] HARDY, N. KeyKOS architecture. *SIGOPS Operating Systems Review* (1985).

[28] HARRIS, T. Hardware trends: Challenges and opportunities in distributed computing. *ACM SIGACT News* (2015).

[29] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., AND SCHÖNBERG, S. The performance of μ-kernel-based systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles - (SOSP)* (1997).

[30] HEISER, G., AND ELPHINSTONE, K. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)* (2016).

[31] HP LABS. The Machine. https://www.labs.hpe.com/the-machine, 2018. Accessed: May, 2018.

[32] JÄRVINEN, K., AND SKYTTÄ, J. High-speed elliptic curve cryptography accelerator for Koblitz curves. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2008).

[33] KAESTLE, S., ACHERMANN, R., HAECKI, R., HOFFMANN, M., RAMOS, S., AND ROSCOE, T. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2016).

[34] KARNAGEL, T., HABICH, D., AND LEHNER, W. Adaptive work placement for query processing on heterogeneous computing resources. In *Proceedings of Very Large Data Bases (VLDB)* (2017).

[35] KIM, S., HUH, S., ZHANG, X., HU, Y., WATED, A., WITCHEL, E., AND SILBERSTEIN, M. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[36] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)* (2009).

[37] KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A., AND UHLIG, V. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys)* (2006).

[38] KRILL, B., AMIRA, A., AND RABAH, H. Generic virtual filesystems for reconfigurable devices. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)* (2012).

[39] KUMAR, A. Intel's new mesh architecture: The "superhighway" of the data center – IT Peer Network, 2017.

[40] LACKORZYNSKI, A., AND WARG, A. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems (IIES)* (2009).

[41] LIEDTKE, J. On µ-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (OSDI)* (1995).

[42] LIN, F. X., WANG, Z., AND ZHONG, L. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2014).

[43] LIU, D., CHEN, T., LIU, S., ZHOU, J., ZHOU, S., TEMAN, O., FENG, X., ZHOU, X., AND CHEN, Y. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[44] LYONS, A., MCLEOD, K., ALMATARY, H., AND HEISER, G. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (2018).

[45] MARTIN, M. M. K., HILL, M. D., AND SORIN, D. J. Why on-chip cache coherence is here to stay. *Communications of the ACM (CACM)* (2012).

[46] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM (CACM)* (1960).

[47] MCKENNEY, P., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. *Ottawa Linux Symposium (OLS)* (2001).

[48] MILLER, M. S., YEE, K.-P., SHAPIRO, J., ET AL. Capability myths demolished. Tech. rep., Johns Hopkins University Systems Research Laboratory, 2003.

[49] NEEDHAM, R. M., AND WALKER, R. D. The cambridge CAP computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP)* (1977).

[50] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., AND HUNT, G. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)* (2009), SOSP.

[51] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[52] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. In *Proceedings of Computing Systems, Volume 8* (1995).

[53] REESE, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[54] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. Ptask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[55] SHAN, Y., HUANG, Y., CHEN, Y., ZHANG, Y., AND OSDI, I. LegoOS : A disseminated , distributed OS for hardware resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2018).

[56] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)* (1999).

[57] SILBERSTEIN, M. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[58] SILBERSTEIN, M., FORD, B., KEIDAR, I., AND WITCHEL, E. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[59] SO, H. K.-H., AND BRODERSEN, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transaction of Embedded Computing Systems (TECS)* (2008).

[60] SODANI, A. Knights landing (KNL): 2nd generation Intel® Xeon Phi processor. In *Proceedings of Hot Chips 27 Symposium (HCS)* (2015).

[61] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with OS clustering. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)* (2011).

[62] STEINBERG, U., AND KAUER, B. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2010), ACM, pp. 209–222.

[63] TANENBAUM, A., MULLENDER, S., AND RENESSE, R. V. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)* (1986).

[64] WATSON, R. N., WOODRUFF, J., NEUMANN, P. G., MOORE, S. W., ANDERSON, J., CHISNALL, D., DAVE, N., DAVIS, B., GUDKA, K., LAURIE, B., MURDOCH, S. J., NORTON, R., ROE, M., SON, S., AND VADERA, M. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[65] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium (USENIX Security)* (2010).

[66] WENTZLAFF, D., AND AGARWAL, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review* (2009).

[67] WOODRUFF, J., WATSON, R. N. M., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014).

[68] ZELLWEGER, G., GERBER, S., KOURTIS, K., AND ROSCOE, T. Decoupling cores, kernels, and operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

Pragh: Locality-preserving Graph Traversal with Split Live Migration

Xiating Xie, Xingda Wei, Rong Chen, Haibo Chen
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: rongchen@sjtu.edu.cn

Abstract

Many real-world data like social, transportation, biology, and communication data can be efficiently modeled as a graph. Hence, graph traversal such as multi-hop or graph-walking queries has been key operations atop graph stores. However, since different graph traversals may touch different sets of data, it is hard or even impossible to have a one-size-fits-all graph partitioning algorithm that preserves access locality for various graph traversal workloads. Meanwhile, prior shard-based migration faces a dilemma such that coarse-grained migration may incur more migration overhead over increased locality benefits, while fine-grained migration usually requires excessive metadata and incurs non-trivial maintenance cost.

This paper proposes Pragh, an efficient locality-preserving live graph migration scheme for graph store in the form of key-value pairs. The key idea of Pragh is a split migration model which only migrates values physically while retains keys in the initial location. This allows fine-grained migration while avoiding the need to maintain excessive metadata. Pragh integrates an RDMA-friendly location cache from DrTM-KV to provide fully-localized accesses to migrated data and further makes a novel reuse of the cache replacement policy for lightweight monitoring. Pragh further supports evolving graphs through a check-and-forward mechanism to resolve the conflict between updates and migration of graph data. Evaluations on an 8-node RDMA-capable cluster using a representative graph traversal benchmark show that Pragh can increase the throughput by up to 19× and decrease the median latency by up to 94%, thanks to split live migration that eliminates 97% remote accesses. A port of split live migration to Wukong with up to 2.53× throughput improvement further confirms the effectiveness and generality of Pragh.

1 Introduction

Graph data ubiquitously exist in a wide range of application domains, including social networks, road maps, biological networks, communication networks, electronic payment, semantic webs, just to name a few examples [47]. Graph traversal (aka multi-hop or graph-walking) queries have been prevalent and important operations atop graph store to support emerging applications like fraud detection in e-commerce transaction [45], user profiling in social networking [11, 18, 6], query answering in knowledge base [52, 63], and urban monitoring in smart city [64].

With the increasing scale of data volume and the growing number of concurrent operations, running graph traversal workloads over distributed graph store becomes essential. Graph traversal workloads are severely sensitive to the access locality, while it is notoriously difficult to partition graph with good locality. For example, the difference of median latency for two-hop query (like friends-of-friends (FoF) [18]) over a Graph500 dataset (RMAT26) [12] is about 30× (0.75ms vs. 22.5ms) between a single machine and an 8-node cluster. Further, preserving locality is even more challenging where workloads and datasets may evolve, while it is common for many production applications [7, 16, 42, 33].

We argue that live migration of graph data is a necessary mechanism for preserving access locality in graph traversals, because existing alternatives have several limitations in many scenarios. First, locality-aware graph partitioning algorithms may improve the performance of a specific dataset and workload [27, 13]. However, *one partition scheme cannot fit all* [62]. Further, a proper graph partitioning scheme for a certain workload may be ineffective and even harmful to another graph traversal workload. Second, replicating data to multiple or all machines allows more (fast) localized read accesses, but also leads to excessive memory overhead as the increase of machines and heavy synchronization cost among replicas for write operations.

Hence, live migration becomes a compelling approach to preserve locality, which has been widely investigated in the database and distributed systems community over the last decade. Unfortunately, the unique characteristics of graph data and traversal operations significantly weaken the benefits of live migration using a shard-based approach, even which is adopted by almost all existing systems. For example,

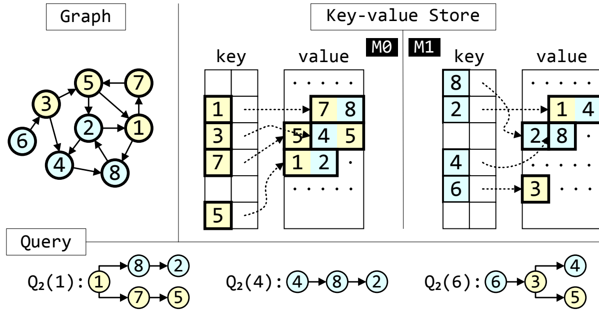


Fig. 1. A sample graph (G), key-value store over 2 machines, and three two-hop queries (Q_2).

using a typical shard-based live migration [25, 60] with an optimal migration plan on above two-hop query experiment will just decrease 29% (22.5ms vs. 15.9ms) median latency, still far from the performance of ideal setting (pure localized access). This is because the majority of the migrated data in a shard would likely have different location preferences. On the other hand, decreasing the size of the shard (fine-grained) would incur high memory and CPU overhead due to storing and maintaining excessive metadata (a location mapping for every shard).

In this paper, we present Pragh, an efficient locality-preserving live migration scheme for distributed in-memory graph store. The key idea of Pragh is a new migration scheme called *split live migration*, which separates the migration of keys and values. Only the value would be migrated physically, while the key would always be stationary at its initial location. This allows fine-grained migration (vertex granularity) while avoiding the need to maintain excessive metadata.

Pragh is made efficient and cost-effective with several key design choices. First, to migrate well-selected vertices (scattered over the entire store) efficiently, Pragh proposes a *unilateral migration protocol* such that the target machine can migrate vertices alone by carefully leveraging one-sided RDMA primitives, while the traversal workloads can concurrently execute on the store. Second, Pragh integrates split live migration with location-based caching [61] to provide *fully-localized* accesses to migrated data. This eliminates the restriction from the stationary key and unleashes the full power of split migration. Third, to support the evolving graph with live migration, Pragh designs a *check-and-forward mechanism* to resolve the conflict between updating and migrating data. Finally, fine-grained monitoring both local and remote accesses to every vertex may incur non-trivial memory and CPU overhead to traversal workloads. Pragh makes a novel reuse of the cache replacement policy to concentrate on tracking remote data accessed frequently. Pragh further provides two optional mechanisms (eager and deferred) for local access tracking to balance the accuracy and the timeliness of migration.

We have implemented Pragh by extending DrTM-KV [61], a state-of-the-art RDMA-enable key-value store, to store graph data and support split live migration. To demonstrate

Table 1: A detail analysis of shard-based live migration.

	Ideal	Shard-based	
		Before	After
Throughput (K ops/sec)	3,248	123	171
Median/50th Latency (msec)	0.75	22.5	15.9
Tail/99th Latency (msec)	4.2	76.6	59.2
Remote Access Rate (%)	0	86.2	64.4
Data Migration Rate (%)	-	-	85.6

the effectiveness and efficiency of Pragh, we have conducted a set of experiments using a state-of-the-art graph traversal benchmark on an 8-node RDMA-capable cluster. The experimental results show that Pragh can increase the throughput by up to 19× and decrease the median latency by up to 94% through live migration, as the rate of remote accessing reduces from 86.2% to 2.0%. We have also integrated split live migration to Wukong [52], a state-of-the-art distributed graph store that leverages RDMA-based graph exploration (graph traversals in parallel) to provide highly concurrent and low-latency queries. An evaluation using original concurrent workload benchmark [52] shows that the throughput increases by up to 2.53× due to using split live migration.

2 Background and Motivation

2.1 Graph Store and Traversal Workload

The graph-structured store (aka graph store) becomes more and more prevalent in an increasing number of applications [47] for modeling the relationships among connected data. Due to fast lookup and good scalability, distributed key-value stores are widely used by existing graph systems [52, 64, 57, 22, 31, 24, 51, 63, 54] as the underlying storage layer to support graph traversal operations efficiently, which play a vital role for many emerging and crucial applications [45, 11, 18, 63, 52].

A natural way to build a graph model on top of the key-value store is to simply use the vertex as the key and the adjacency list as the value [51]. Further, separate key and value memory regions are used to support variable-sized key-value pairs [39, 61, 52]. Specifically, the key region is a fix-sized hash table, where each entry stores a key and an address (i.e., offset and size) of the value region. The value region stores variable-sized values consecutively. As shown in Fig. 1, a sample graph (G) is stored into a key-value store over two machines. Various graph traversal operations (like FoF, multi-hop query, and random walking) can be implemented by iteratively accessing key and value pairs. For example, the two-hop query on vertex 1 ($Q_2(1)$) will first retrieve neighbors of the start point (vertex 1) by hashing it as the key and accessing its value (vertex 7 and 8). The next hop will use the value in this hop as the keys (hash(7) and hash(8)) to retrieve their neighbors (vertex 2 and 5) recursively. The accesses over key and value may be either local or remote according to the partitioning scheme.

2.2 Poor Locality and Partitioning

For distributed in-memory stores, the locality of data accessing is quite important because accessing local memory is still more than 20× faster than accessing remote memory across networks, even using high-speed networks [22]. Unfortunately, the traversal on distributed graph data is notoriously slow due to poor data locality. Prior work shows that assigning vertices to N machines randomly will lead to the expected fraction of remote accesses reaching $1 - \frac{1}{N}$ [27].

To illustrate the performance impact of locality for graph store, we conducted a motivating experiment using two-hop queries (like FoF [18]) over Graph500 dataset [12] (RMAT26) on an 8-node RDMA-capable cluster. The graph is partitioned into 8 machines randomly (hash-based), and a set of vertices randomly sampled with a Zipf distribution (θ=0.99) is used to run two-hop queries which access fixed 100 friends of 100 friends. As shown in Tab. 1, the distributed setting using 8 machines only achieves less than 4% throughput (123K vs. 3,248K) and about 30× median (50th percentile) latency (22.5ms vs. 0.75ms) of the ideal setting since the rate of remote accessing reaches up to 86.2%.[1]

Therefore, designing locality-aware graph partitioning algorithms has been an active area of research for a decade [27, 13], especially for graph analytics systems. However, *one partition scheme cannot fit all* [62]. It is hard or even impossible to handle dynamic workloads or evolving graphs only relying on static partition-based approaches. One example is shown in Fig. 1 such that $Q_2(1)$ and $Q_2(4)$ contends for the same vertices (8 and 2). The queries may arrive at different times, which causes *false contention*. Actually, prior work on production applications has shown that workloads change rapidly over time [7, 16, 42, 33].

2.3 Live Migration

Live migration (aka dynamic migration) is a compelling approach for handling dynamic workloads and has been widely investigated in the database and distributed systems communities [20, 21, 26, 25, 35, 60, 5]. Generally, a centralized coordinator will make the migration plan according to the statistics (e.g., access frequency) collected by the monitor on each machine. The migration threads on the source and/or target machines will implement the plan by migrating key-value pairs in a synchronous way (see Fig. 2(b)). Since the position of vertices may change after migration, additional metadata (POS) will be accessed to look up the latest positions of the key-value pairs before accessing them (see Fig. 2(a)). The metadata should be updated by the coordinator during live migration and usually is consistently cached at each machine to avoid remote lookup for every accesses.

[1]The ideal result is gained by running the benchmark on a single machine (fully local accessing). The throughput is further magnified 8× (the number of machines).

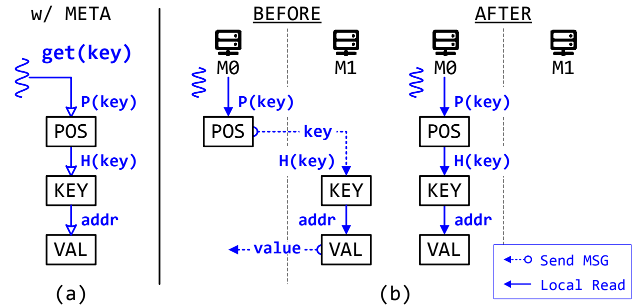


Fig. 2. (a) The sequence of an access on (kv-based) graph store and (b) a comparison of accesses before and after live migration.

Shard-based migration. A ubiquitous approach in live migration is to group the data into *shards* (*partitions*) by key ranges or key hashing [49, 53, 4, 35, 5]. Shards serve as the unit of migration for load balancing and locality-aware optimization. Prior work mainly focuses on relational workloads (e.g., TPC-C) or simple CRUD (Create, Read, Update, and Delete) workloads (e.g., YCSB [15]). Compared to traversing graph data, such workloads with datasets usually have high access locality (e.g., accessing 1% remote key in TPC-C). Consequently, leveraging shard-based migration on the graph store is ineffective and may be harmful, due to the following reasons:

First, migrating data at shard granularity will significantly weaken the benefits of data migration. Due to lacks of data locality, after migrating a shard, the majority of the migrated data in the shard would likely not be accessed by the workload at the target machine. Meanwhile, it will also increase the number of remote accesses at the source machine. Based on the above motivating experiment, we partition graph data into one hundred shards per machine (about 70K keys per shard), similar to prior work [11, 5]. All of the local and remote accesses to every shard are monitored and aggregated to make an optimal migration plan. As shown in Tab. 1, the rate of remote accessing only decrease from 86.2% to 64.4% even after migrating more than 85.6% of graph data (about 20GB). As a result, the throughput only increases 39% (123K vs. 171K) and the median latency also just decreases 29% (22.5ms vs. 15.9ms), still far from the performance of an ideal setting.

Second, though decreasing the size of shard could enhance the effectiveness of migration, it still faces the same drawbacks of static graph partitioning approaches when handling dynamic workloads, unless vertices (key-value pairs) serve as the unit of migration. For example, two irrelevant queries may contend the same shard even assigning two vertices to one shard by key ranges, like vertex 2 and 4 for $Q_2(1)$ and $Q_2(6)$ in Fig. 1. More importantly, the amount of metadata (POS) needed to manage the shards would incur extremely high memory pressure. For example, the metadata for the motivating experiment will consume about 3GB memory on each machine to support vertex granularity migration. Each machine has to cache the entire metadata since the workload

may access any vertex of the graph. Consequently, the size of metadata may exceed the size of graph data when the graph scales.

3 Approach and Overview

Our approach: split live migration. We propose a new migration approach, named *split live migration*, that enables live migration at the minimum level of granularity (i.e., key-value pair). A landmark difference compared to prior approaches is that *split live migration has no need of metadata at all*. This is the greatest advantage but also the biggest challenges for live migration.

The key principle of split migration is to separate the migration of keys and values. The key will always be *stationary* at its initial location, which can be found without metadata (e.g., key hashing). The value will be *migratory* on demand to improve locality or rebalance the load. Our design naturally tackles the issue of memory pressure by avoiding metadata due to the stationary key. Further, allowing fine-grained migration (even a single value) would maximize the effectiveness of data migration for graph store. However, there are still many challenges before making split live migration come true.

Opportunity: RDMA. Remote Direct Memory Access (RDMA) is a networking feature to provide cross-machine accesses with high speed, low latency, and kernel bypassing. The one-sided RDMA primitive (e.g., `READ`, `WRITE`, and `CAS`) allows one machine to directly access the memory of another machine without involving the host CPU. Much prior work has demonstrated the benefit of using RDMA for in-memory key-value stores [39, 22, 32, 61]. Generally, the GET/PUT (read/write) operation first uses `RDMA READ`s to look up the location (address) of the value by hashing the given key, and then use `RDMA READ/WRITE` to retrieve/update the value (see the left part of Fig. 4(b)). We observe that *one-sided RDMA primitives decouple the accesses of keys and values, which make it easy and efficient to separate keys and values in physical.* It opens a new opportunity to *split* live migration.

Challenges and solutions. First, split live migration uses the key-value pair as the unit of migration, such that the key-value pairs which will be migrated are scattered over the entire graph store. Therefore, directly using existing protocols designed for shard-based migration may be inefficient. We propose a unilateral (target-only) migration protocol that the target machine can do it alone and efficiently by carefully leveraging one-sided RDMA primitives (§4.1).

Second, the basic split migration only migrates the values of key-value pairs, which can at most eliminate about half of the remote accesses. This is because the read access to the key of key-value pair (look up the location of the value) will still be remote. We address this challenge by integrating split migration with RDMA-friendly location-based caching [61] to provide *fully-localized* access to migrated data (§4.2).

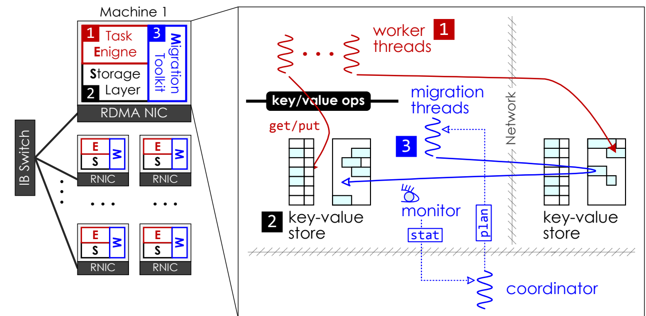


Fig. 3. The architecture of Pragh.

Third, the split of key and value after performing migration presents a new challenge to the support of evolving graphs, especially for the target-only protocol. We use a check-and-forward mechanism to resolve the conflict between data updating and data migrating tasks (§4.3).

Finally, to maximize the effectiveness of data migration, both local and remote accesses to every key-value pair should be tracked to generate an optimal migration plan. It may incur non-trivial memory and CPU overhead to traversal workloads. We design a lightweight, memory-saving monitor, which reuses the location cache to track frequent remote accesses and provides two optional mechanisms for local access tracking to balance the accuracy and the timeliness of live migration (§4.4).

Architecture. As shown in Fig. 3, Pragh is a distributed in-memory graph store with split live migration. It follows a decentralized architecture to deploy servers on a cluster of machines connected with a high-speed, low-latency RDMA network. Each server is composed of three components: task engines, a storage layer, and a migration toolkit. The task engine binds a worker thread on each core with a task queue to continuously execute operations (e.g., GET and PUT) from clients or other servers. The storage layer adopts an RDMA-enabled key-value store over distributed hashtable to support a partitioned global address space. The migration toolkit enables a monitor to collect statistics of graph store and runs migration threads to perform live migration. Pragh scales by partitioning graph data randomly (hash-based) into multiple servers. Each server stores a partition of the graph, which is shared by all of the workers and migration threads on the same machine.

Execution flow. Pragh is designed to handle concurrent operations on graph data with low-latency and high-throughput. The key advantage of Pragh over previous systems is capable of physically migrating data to improve locality in a split way, which can promptly and significantly enhance performance for dynamic workloads.

Similar to prior work [53, 60, 35], a *centralized coordinator* will make migration plan according to the statistics (e.g., access frequency) collected by the monitor on each server and migration policies. The details – how to make a proper policy and how to find an optimal plan – are beyond the

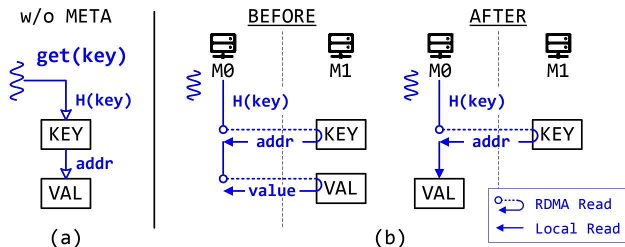


Fig. 4. (a) The sequence of an access on (kv-based) graph store without meta-data and (b) a comparison of accesses before and after split live migration.

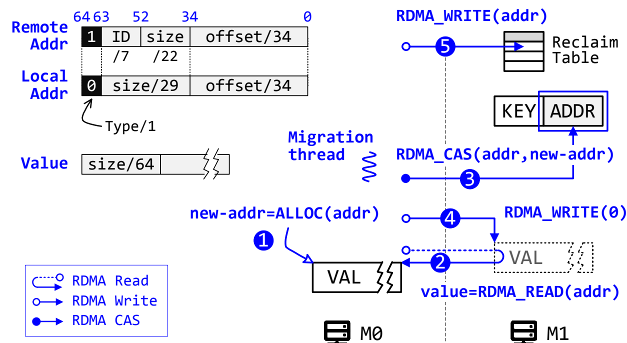


Fig. 5. The execution flow of basic split migration.

scope of this paper and are part of our future work. Currently, Pragh uses a simple threshold-based policy to generate migration plans. On each server, the monitor will track the accesses of worker threads to the key-value store in the background and report to the coordinator periodically (e.g., 10s) or instantly (e.g., when exceeding 100 times per second). The coordinator will compare the statistics from the applicant and the machine hosting the vertex at present, and approve the migration if the profit is more than a threshold (e.g., 50% more accesses per second). After that, the migration threads will migrate the key-value pairs according to the plan from the coordinator, while the worker threads will continue to execute queries by accessing the same key-value store concurrently. Note that the centralized coordinator is just used to collect a few statistics from servers and approve migrations by simply comparing the statistics. Further, the fine-grained approach commonly only needs to migrate much fewer vertices (e.g., 0.13% in §6.1). Hence, the coordinator may hardly become a bottleneck in a medium-sized cluster.

4 Split Live Migration

Pragh uses an RDMA-enabled key-value store over distributed hashtable to store graph data physically. For brevity, Pragh supposes that each vertex has a unique ID (vid) and use it as the key. The hash value of the key (`H(key)`) can be used to identify the host machine (`mid`) and the location in the key region (`off`). As shown in Fig. 4(a), to get neighbors of a given vertex, the worker thread first uses `H(key)` to look up the address of its value and then retrieves the value (a list of IDs of neighbors). For remote key-value pairs, `RDMA READ`s are used to access keys and values (see the left part of Fig. 4(b)), which are at least 20× slower than local reads. Hence, Pragh uses split live migration to eliminate such remote accesses.

4.1 Basic Split Migration

We start from the basic migration protocol, assuming that there only exist traversal workloads (i.e., GET operations) in the graph store. Since the key is always stationary in the split migration, Pragh will only move the value to the target machine. This could improve locality by avoiding remote accesses to the values (see the right part of Fig. 4(b)).

Address layout. To avoid the influence between accessing and migrating key-value pairs, the address (within the key) should be changed from local to remote in a lock-free way (e.g., compare-and-swap (CAS)). Therefore, both the local and remote location of value should use a 64-bit address uniformly, which can be modified atomically using both local and RDMA atomic instructions.[2]

Considering the machine ID should be added into the address, a simple layout may severely restrict the scope of address space. Pragh adopts a differentiated layout for local and remote addresses. As shown in the top left corner of Fig. 5, The most significant bit is used to present the type of address, local (0) or remote (1). For local addresses, the rest of the bits are used to store 29-bit value size and 34-bit offset within the value region. Thus, the size of a single value and value region on a single machine can reach 4GB and 128GB respectively (assuming 8-byte granularity and alignment). For remote addresses, the value offset still occupy 34 bits to present the entire remote value region, while the value size reduces to 22 bits for hosting 7-bit machine ID. Thus, the graph store can scale up to 128 machines, while the size of the maximum value that can be migrated is limited to 32MB. The observation is that the system will prefer to migrate the workloads rather than very large key-value pairs [52, 57]. Further, a large key-value pair can be split into multiple ones (vertex decomposition [52, 57]), and each one can be migrated separately.

Unilateral migration protocol. Similar to traditional shard-based migration systems, the split live migration also could be implemented by the collaboration of migration threads on source and target machines. However, the key-value pairs which will be migrated, are scattered over the entire graph store due to lacks of locality. It means that migrating multiple key-value pairs may incur a prolonged interruption to the concurrent graph accessing and/or lengthy migration delay since multiple addresses (within separated keys) should be modified by atomic operations (e.g., CAS).

Pragh proposes a unilateral (target-only) migration protocol based on one-sided RDMA primitives. It only uses the

[2] Note that RDMA primitives guarantee atomic 64-bit transfer [9], and `RDMA READ/WRITE` operations are also cache coherent with local accesses [22, 61].

```
   MIGRATE(key)
   1 retry:
   2   kmid = H(key).mid          ▸ e.g., key % machines
   3   addr = LOOKUP(kmid, key)
❶  4   buf = ALLOC(addr.sz)
   5   new_addr = { 1, local_mid, addr.sz, buf }
❷  6   RDMA_READ(addr.mid, buf, addr.off, addr.sz)
❸  7   if !RDMA_CAS(kmid, H(key).off, addr, new_addr)
   8   | goto retry                 ▸ conflict w/ PUT
   9   zero = 0                     ▸ invalidate value
❹ 10   RDMA_WRITE(addr.mid, addr.off, zero, 8)
❺ 11   RDMA_WRITE(addr.mid, reclaim, addr, 8)  ▸ reclaim
```

Fig. 6. Pseudo-code of MIGRATE operation.[3]

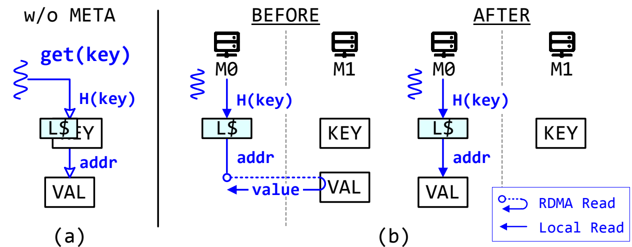


Fig. 7. (a) The sequence of an access on (kv-based) graph store with location cache and (b) a comparison of accesses before and after split live migration with location cache (for remote kv pair).

migration thread on the target machine to migrate the key-value pair instantly, while the worker threads on every machine can still access the key-value pair concurrently. Fig. 5 illustrates three steps of the migration protocol (a detail pseudo-code is shown in Fig. 6). First, the migration thread on the target machine will allocate memory space in local value region (`new_addr`) to host migrated value (❶), according to the size in the original address. Second, the migration thread will retrieve the value using one RDMA READ from the original address to the new address (❷). Finally, one RDMA CAS is used to replace the original (local) address with the new (remote) address (❸).

Invalidation and reclaim. Unilateral migration protocol will incur two new problems. First, the memory of migrated value in the source machine should be invalidated. However, some worker threads may still have the original address of the migrated value and will access it in the future. To solve it, Pragh proposes a passive invalidation mechanism. The migration thread will invalidate the original memory of migrated value by zeroing (RDMA WRITE) the size within the value (❹). Before using the retrieved value, the worker thread should check whether the size within the value and address are equal. If not, the address should be regained. Note that the worker thread can safely read the value from the original memory before invalidation even it has just been migrated (❸).

Second, the memory of migrated value on the source machine should be reclaimed. However, it is hard or even impossible for the migration thread on the target machine to solely free the memory. Therefore, Pragh uses a lease-based mechanism to reclaim the memory of migrated values in the background by a garbage collection (GC) thread on the source machine.[4] The migration thread will actively write (RDMA WRITE) the original address to the reclaim table[5] of the source machine, at the end of live migration (❺). The GC thread on each machine will periodically check the reclaim table to free the expired memory, which has been migrated before a pre-agreed lease (e.g., 60s). All the worker threads comply with the convention that the value address obtained before a lease duration should not be used, since it may have been freed and reused. The performance impact could be trivial by using a long-term lease.

4.2 Fully-localized Split Migration

The basic split migration only avoids remote accesses to the values, which limits the effects of migration since only at most half of remote accesses can be eliminated.

Observation: location cache. Prior work [61, 23, 59] proposes location-based caching for RDMA-friendly key-value stores, which aims at avoiding remote accesses to the keys. Different to caching the content (value) of key-value pairs, the location cache (L$) only stores the location (address) of key-value pairs, which is very space-efficient and effective (see the left part of Fig. 7). We observe that *location cache is a perfect counterpart to split migration*. They focus on two different halves of the access to the remote key-value pair, and the candidates of them are also well matched, namely remote key-value pairs frequently accessed. Finally, a small cache has negligible memory overhead (e.g., 128MB) and lookup cost, yet it is sufficient to achieve fully-localized accesses for most workloads [46, 61].

Integration with location cache. Pragh extends the graph store with location cache (L$) and integrates it with split live migration to enable fully localized accesses after migration. Fig. 8 illustrates the pseudo-code of GET operation with the integration of location cache and split live migration. When accessing a remote key-value pair (Line 9), the worker thread will first check location cache (Line 21) and fill the cache (if missed) with the address of the value (Line 25) obtained by the remote access to the key (Line 24). Given the address, the worker thread will retrieve the value using one RDMA READ (Line 14).

If the worker threads access the key-value pair frequently enough, the value will be migrated to the local using the basic migration protocol. After that, the address stored in location cache will be updated by the new address, which points to the local value region. Therefore, the accesses to the key-value pair will be fully localized (Line 10-12), as shown in the right part of Fig. 7. In contrast, the local key-value pair could also

[3]RDMA provides fences between different requests [38], and Pragh uses them before invalidating and reclaiming the memory (Line 10 and 11).
[4]Pragh uses the precision time protocol (PTP) [1] to implement lease.
[5]We implement the reclaim table like the circular buffer [22].

```
 GET(key, buf)
+1 retry:
 2   kmid = H(key).mid           ▸ e.g., key % machines
 3   addr = LOOKUP(kmid, key)
 4   if kmid == local_mid                    ▸ local key
+5   | if addr.type == 0                   ▸ local value
 6   | | MEMCPY(buf, vals[addr.off], addr.sz)
+7   | else                ▸ remote value (migrated)
+8   | | RDMA_READ(addr.mid, buf, addr.off, addr.sz)
 9   else                                   ▸ remote key
+10  | if addr.type == 1                      ▸ migrated
+11  | | && addr.mid == local_mid         ▸ local value
+12  | MEMCPY(buf, vals[addr.off], addr.sz)
+13  | else                                ▸ remote value
 14  | | RDMA_READ(addr.mid, buf, addr.off, addr.sz)
+15  if CHECK(addr, buf)
+16  | if kmid != local_mid
+17  | | cache.DELETE(key)                   ▸ invalidate
+18  | goto retry

 LOOKUP(kmid, key)
 19  if (kmid == local_mid)                  ▸ local key
 20  |  return keys[H(key).off]
x21  if cache.FIND(key)
+22  |  && !EXPIRED(cache.GET(key).lease)
x23  |  return cache.GET(key).addr            ▸ cache hit
 24  RDMA_READ(kmid, addr, H(key).off, 8)
x25  cache.INSERT(key, addr)                 ▸ fill cache
+26  cache.GET(key).lease = NOW()
 27  return addr
```

Fig. 8. Pseudo-code of GET operation with location cache. The code lines with "x" and "+" stand for additional instructions to integrate with location cache and split live migration, respectively.

be migrated to other machines, thus the type of address will be used to decide how to retrieve the value (Line 5-8).

Finally, the address stored in the location cache should also follow the convention of the invalidation and the reclaim mechanisms. First, if the retrieved value is invalid (Line 15), the worker thread has to delete the address in location cache for the remote key-value pair (Line 16-17), and needs to retry (Line 18). Second, the cached address must expire after a lease duration (e.g., 60s) from the last cache time (Line 22 and 26). Note that the duration of the (cache) lease should be equal or smaller than that of the (reclaim) lease (§4.1).

4.3 Full-fledged Split Migration

The basic migration protocol only considers traversal workloads (i.e., GET operations) concurrently execute in the graph store. Pragh extends it with a check-and-forward mechanism to support the evolving graph (i.e., PUT operations). For brevity, suppose that graph store has provided some mechanisms (e.g., snapshot read [52, 64]) to run traversal workloads over evolving graphs correctly.[6] Therefore, Pragh only tackles the conflict between split live migration and the change of graph. More specifically, Pragh only needs to consider the concurrent update to edges (i.e., change the value of a key-value pair).

[6]Pragh assumes the PUT operation will use atomic in-place updates on the key to ensure consistency, which is common in prior work [52, 64].

```
 PUT(key, val)
+1 retry:
 2   kmid = H(key).mid
 3   addr = LOOKUP(kmid, key)
+4   if addr.mid != local_mid               ▸ migrated
+5   | SEND(addr.mid, key, val)       ▸ forward PUT op
+6   | return false
 7   new_addr = WRITE_VALUE(addr, val)
 8   if !RDMA_CAS(kmid, H(key).off, addr, new_addr)
 9   | goto retry         ▸ conflict w/ put or migrate
 10  zero = 0                         ▸ invalidate value
 11  MEMCPY(vals[addr.off], zero, 8)
 12  MEMCPY(reclaim, addr, 8)                ▸ reclaim
 13  return true
```

Fig. 9. Pseudo-code of PUT operation. The code lines with "+" stand for additional instructions to support split live migration.

We observe that both MIGRATE and PUT operations will change the address within the key atomically to mark the success of processing (Line 7 in Fig. 6 and Line 8 in Fig. 9).[7] Moreover, PUT operation will always be assigned to the machine hosting the key at first. So for key-value pairs migrated, a better choice is to forward the PUT operation to the machine hosting the value upon conflicts, which also ensures consistency and reclaims the memory. Consequently, Pragh adopts different strategies for MIGRATE and PUT operations when detecting the conflict over the address; MIGRATE operation will be retried (Line 8 in Fig. 6), while PUT operation will forward itself (Line 5 in Fig. 9), if it conflicts with some MIGRATE operation and then is retried (Line 9 in Fig. 9). Note that PUT operation will always update the address in the machine hosting the key using RDMA CAS, even though PUT operation is forwarded.

4.4 Lightweight Monitoring

To generate a proper migration plan, the coordinator should collect the statistics of both local and remote accesses to every key-value pair. A (much) higher remote access number from a certain machine to a key-value pair in the most recent interval (e.g., 10s) indicates that migrating the key-value pair to that machine may improve locality (fewer remote accesses). It has been a great challenge to track the accesses at the granularity of key-value pairs.[8] Even worse, the remote accesses using RDMA READ contributes much more extra burdens (both memory and CPU overhead) to the monitor, since each machine has to track the accesses to remote key-value pairs (except local key-value pairs).

Pragh designs a lightweight, memory-saving monitor for split live migration by tracking local and remote accesses separately. For remote accesses, worker threads may access

[7]Suppose that PUT operation will change the size or the offset of the address (or both), namely addr is not equal to new_addr.

[8]Relational database can leverage table schema to reduce the number of tuples should be tracked, by grouping co-accessed tuples into blocks [53, 25, 50, 60]. Unfortunately, graph store is generally schema-less.

any key-value pairs, while the monitor may (very likely) only care about remote key-value pairs *accessed frequently*. This observation also matches the intention of the location cache. Hence, Pragh reuses the cache to track (partial) remote accesses(remote key). The monitor relies on the replacement policy of cache to recognize the key-value pairs (worth tracking) freely. Note that the accesses for the values migrated to local will still be tracked through the cache.

For local accesses, reserving space for every key and tracking every access might be not worth, especially for a very large store. This is because only a small fraction of key-value pairs should be migrated for a while. For example, migrating less than 0.2% of key-value pairs is sufficient for the motivating experiment (§6.1). Therefore, Pragh allows to skip tracking local accesses to the key-value pairs and provides two optional mechanisms to balance the timeliness and the accuracy of split live migration. Note that the monitor on each machine will report to the coordinator when remote accesses to a key-value pair exceed a threshold.

Eager migration: The coordinator will eagerly approve the migration of the key-value pair. After migration, the machine hosting the key will track the (remote) accesses to the key-value pair using a separate table, and then may migrate it back in future if it accesses the key-value pair more frequently.[9]

Deferred migration: The coordinator will notify the machine hosting the key to track the (local) accesses to the key-value pair using a separate table. After a migration interval, the coordinator will decide whether to migrate the key-value pair according to the statistics from all of the machines.

4.5 Discussion

Even though the current design of split live migration highly relies on RDMA, we believe that it can still benefit graph traversal workloads without RDMA, including no need for metadata and vertex granularity migration. However, after migrating the value to local, the cost to retrieve the address would be almost the same as the cost to retrieve the value directly. Hence, location cache must be deployed even without RDMA. On the other hand, the lack of RDMA would also need to rethink the implementation of migration protocol. Our future work may extend Pragh to support commodity networks without RDMA.

5 Implementation

Fault tolerance. Pragh supposes distributed in-memory graph store has provided durability and/or availability by using specific mechanisms like checkpointing or replication. Pragh only needs to consider the interrupted migration tasks and the recovery of crashed machines, because split live migration only changes the location of key-value pairs rather than the content of key-value pairs.

Interrupted migration tasks: If the crashed machine is the source of migration, there is nothing to do since the key-value pair will be recovered on the crashed machine later. If the crashed machine is the target of migration, a corner case that the interruption occurs after replacing address (❸ in Fig. 5) but before reclaiming memory (❺ in Fig. 5) will cause a little memory leakage, which can be detected and reclaimed by scanning the entire value memory region in background.

System recovery: Pragh relies on the mechanism provided by graph store to detect machine failures, like Zookeeper [30]. It will notify surviving machines to assist the recovery of crashed machines, which needs to handle two kinds of key-value pairs. First, the key-value pairs hosted by a crashed machine will be reloaded by the substitute of the crashed machine. Before that, all surviving machines will flush addresses in location cache which point to the key-value pairs hosted by crashed machines (i.e., `H(key).mid`) whether they have been migrated or not, and reclaim the memory of values migrated from crashed machines. Second, the key-value pairs, hosted by a surviving machine but migrated to a crashed machine, will be reloaded by the surviving machine. Before that, all surviving machines will also flush addresses in location cache which point to the key-value pairs migrated to the crashed machines (i.e., `addr.mid`). The coordinator will record the latest target machines of values migrated persistently before approving the migration, which could help surviving machines reload vertices precisely. Moreover, all workloads running on surviving machines involving crash machines will be aborted and suspended until recovery is complete. Finally, the coordinator in Pragh is stateless and easy to recover. The coordinator failure will not influence the execution of worker threads and only pause launching new migration tasks and recovering crashed machines. The migration thread can continue to complete the outstanding migrations.

Optimizations. Pragh adopts a unilateral migration protocol (see §4.1) to migrate one key-value pair (vertex) at a time, which requires *at most* five one-sided RDMA operations: two `READ`s to lookup and retrieve the value, one `CAS` to change the address atomically, and two `WRITE`s to invalidate and reclaim the original memory. Though this approach can provide instant response to migration demands and fully bypass the CPU and kernel of source machine, the throughput of migration may be bottlenecked by the network due to too many RDMA operations with small payloads.

To remedy it, Pragh enables three optimizations to further accelerate split migration. First, in most cases, the migrated key-value pair is frequently accessed by the target machine; thus, its address (very likely) has been already cached in the location cache. It means that the migration thread can skip the first `RDMA READ` to look up the address. Second, Pragh will

[9] Pragh relies on the coordinator to prevent the “ping-pong” of migrations, which prefers not to migrate the vertex competed by multiple machines.

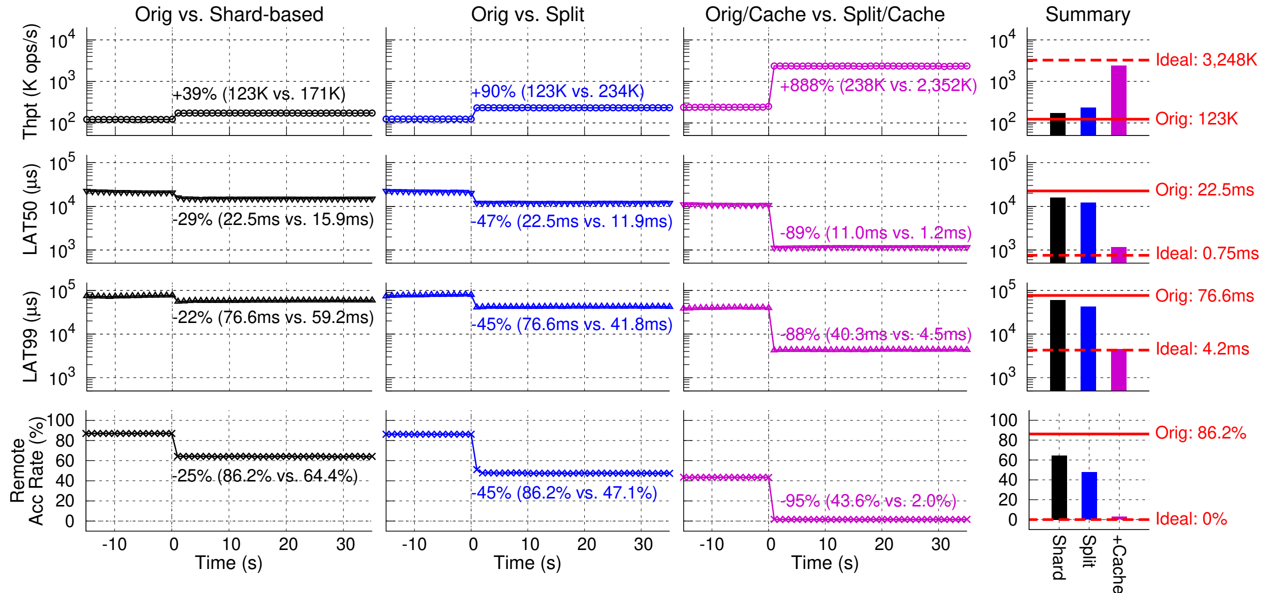


Fig. 10. *A comparison of migration schemes on the traversal benchmark with a skewed workload (a Zipf distribution with $\theta = 0.99$).*

migrate multiple key-value pairs concurrently in a pipelined fashion to better utilize network bandwidth. Each RDMA operation to migrate one key-value pair is implemented as one stage, and Pragh schedules these stages without waiting for the request completion. Finally, since the memory invalidation and reclaim are not on the critical path to migrate one key-value pair[10], Pragh enables passive ACK [59] to acknowledge the completion of such two `RDMA WRITE`s passively, which further reduces the network bandwidth. As a result, a single migration thread is sufficient to migrate more than one million vertices per second (§6).

Load balance. Though Pragh mainly focuses on using live migration to improve the locality of graph traversal workloads, it also can be used to rebalance load across machines, similar to prior work [53, 60, 35]. Basically, it all depends on the migration plan generated by the coordinator. Generally, the traversal workload will be sent to the machine hosting the initial vertex and run to completion. The remote key-value pairs will be retrieved by RDMA operations. Therefore, the coordinator should recognize such hotspots and generate proper plans to scatter them over all of the machines using live migration, like Pragh. Meanwhile, different goals also need different migration policies and statistics. It is orthogonal to the design of Pragh and beyond the scope of this paper.

6 Evaluations

Hardware configuration. All evaluations were conducted on a rack-scale cluster with 8 nodes. Each node has two 12-core Intel Xeon E5-2650 v4 processors and 128GB DRAM. Each node is equipped with two ConnectX-4 MCX455A 100Gbps InfiniBand NIC via PCIe 3.0 x16 connected to a Mellanox SB7890 100Gbps IB Switch, and an Intel X540 10GbE NIC connected to a Force10 S4810P 10GbE Switch. In all experiments, we reserve four cores on each CPU to generate requests to avoid the impact of networking between clients and servers as done in prior work [56, 58, 61, 14, 60, 52]. All experimental results are the average of five runs.

Traversal benchmark. Inspired by YCSB [15], we build a simple benchmark to evaluate the effectiveness of different migration approaches for graph traversal workloads. The traversal benchmark uses a synthetic graph provided by Graph500 [12]. In this paper, the graph with 2^{26} vertices and 2^{30} edges (RMAT26) is used as default dataset since we need to run the benchmark on a single machine to gain the performance of ideal setting (pure localized access). Note that the experimental results on larger graphs (e.g., RMAT29) are similar. The traversal benchmark consists of 95% two-hop queries (Get) and 5% edge updates/inserts (Put), similar to YCSB-B (read-heavy) [15]. Note that the majority of many traversal workloads [11] are two-hop queries, and it is easy to compose other complicated queries like SPARQL query [52]. The starting vertices of two-hop queries are chosen according to a Zipf distribution with $\theta = 0.99$. The scope of starting vertices and the number of neighboring vertices retrieved could be configured. The default values are 2^{10} and 100, respectively. We will compare the performance impact with different settings in separate experiments.

Comparing targets. The following five results are provided in the evaluation of the traversal benchmark. ***Orig*** indicates the performance of running the benchmark over the graph data partitioned randomly and without data migration. ***Ideal*** is the result gained by running the benchmark on a single machine. Specifically, throughput is simply magnified by

[10] To ensure consistency, the (original) memory invalidation must be completed before the next Put operation on (new) memory starts, which is easy to implement with the check-and-forward mechanism (§4.3).

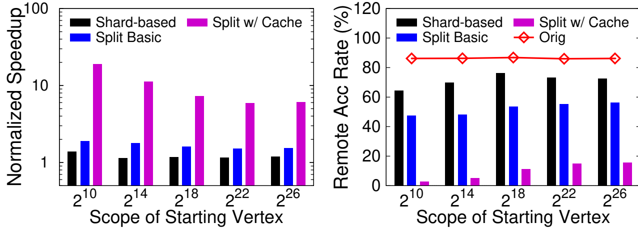


Fig. 11. *A comparison of migration benefits for different approaches with the increase of scopes of starting vertices.*

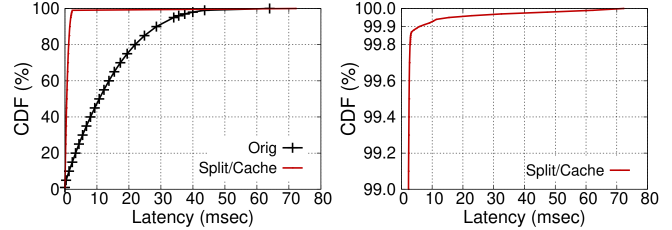


Fig. 12. *The CDF graph of latency for* Put *operations.*

the number of machines (i.e., 8×). ***Shard-based*** represents the performance of a shard-based migration approach, which deploys one hundred shards at each machine, similar to prior work [11, 5]. Note that we always generate optimal migration plans for shard-based migration by tracking every access but do not consider the tracking cost. ***Split/Cache*** and ***Split*** are the performance of Pragh using split live migration with and without location cache. The size of the location cache is 128MB. The migration plan is built by the statistics collected by our lightweight monitor. The default interval is set to 10 seconds.

6.1 Migration Benefits

To study the benefits of migration approaches, we run the traversal benchmark using different migration schemes and compare to the result of the original and ideal settings. As shown in Fig. 10, the original throughput and latency are about 26× slower than the ideal results (123K vs. 3,248K) since about 86.2% accesses to the key-value store are remote. Shard-based approach can only increase the throughput by 39% (123K vs. 171K) and decrease the median (50th percentile) latency by 29% (22.5ms vs. 15.9ms) as it just removes about 25% remote accesses. Pragh can almost double the throughput and reduce the latency by half, thanks to the basic split migration, which removes nearly all remote accesses to the values. Using location cache can remove almost all of the remote accesses to the keys, as the cache hit rate is about 99%. Note that the performance of enabling basic split migration or location cache alone are similar, because both of them still need one `RDMA READ` to retrieve the remote key or value separately.

When combining two techniques, the throughput of Split/Cache can reach 2,352K queries per second (19× compare to Orig). It has achieved close to 72% of ideal performance. The remaining 2.0% of remote accesses is due to the competition on vertices shared by multiple queries running on different machines. Note that traditional migration scheme is hard to integrate with location cache, since they will migrate both keys and values physically and make location cache useless.

Migration time and network traffic. Both split migration and shard-based migration can complete migration in seconds since we optimize the data transmissions in both methods. For shard-based migration, we migrate the shards in block granularity to fully utilize network bandwidth. However, split migration is still faster even using more network round-trips for one key. This is because fine-grained migration migrates much less data than coarse-grained, shard-based migration. For this experiment, only 78,242 keys (0.13% of the total vertices) are migrated in split migration, where 782MB data is migrated in total. For comparison, 85.6% of shards (685 out of 800) are migrated in shard-based migration with a total size of 20GB key-value to transfer.

Scope of starting vertices. Generally, the query will start from a certain type of vertices (e.g., users or tweets in social networks), and the size of the subset of vertices may be various. Fig. 11 further presents the impact of using different scopes of starting vertices in the traversal benchmark from 2^{10} to 2^{26}. The speedup after migration decreases with the increase of scope steadily due to the increase of contention on key-value pairs accessed by multiple queries. It will also result in the rise of remote accesses (see Fig. 11(b)).

Impact on Put operations. To reveal the impact of check-and-forward mechanism in Pragh on the latency of Put operations, we use an update-heavy traversal benchmark, which consists of 50% two-hop queries (Get) and 50% edge updates/inserts (Put), similar to YCSB-A [15]. Fig. 12(a) shows the CDF graph of latency for Put operations with and without split live migration. After migration, the latency of 99.9% Put operations decreases significantly, thanks to the decline of waiting time in the queue. Moreover, as shown in Fig. 12(b), the check-and-forward mechanism will just impact the 99.9^{th} percentile latency, since about 0.11% Put operations updates migrated key-value pairs and is forwarded to another machine. Note that Pragh only migrates 0.13% of total key-value pairs. The increase of latency is mainly contributed by the extra cost for forwarding the operation, waiting in the queue, and re-executing the operation.

Uniform workload. We also evaluate the traversal benchmark with a uniform workload. Shard-based approach can hardly gain benefits and only increases the throughput by 8% (126K vs. 136K) after migration, as the remote access rate just drops from 85% to 82%. In contrast, the basic split migration eliminates over 43% of remote accesses and increases the throughput by 84% (126K vs. 232K). By using location cache, the throughput of Split/Cache can reach 1,521K queries per second (12× compared to Orig). The remote access rate reduces to 5%.

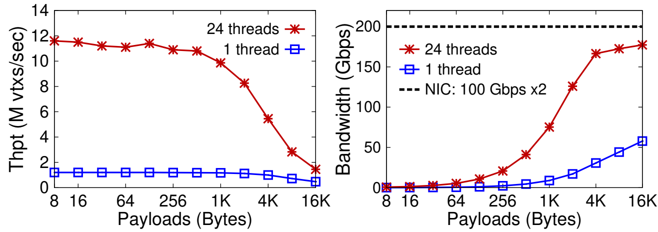


Fig. 13. *The throughput and bandwidth of unilateral migration using 1 and 24 threads.*

6.2 Migration Speed

To evaluate the capability of unilateral migration protocol, we conduct an experiment to migrate values from a remote machine to local with full speed. Fig. 13 shows the throughput of migration and network bandwidth consumed with the increase of payload (i.e., value) size. A single thread is enough to migrate values for millions of vertices per second with less than 4KB payloads. Using parallel migration with 24 threads can further increase the throughput of moving values to more than 10 million per second. Further, using multiple RDMA primitives to migrate a single value will not be limited by network. It should be noted that split live migration will only use the CPU of the target machine.

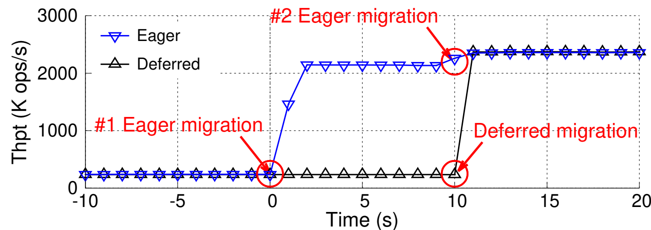


Fig. 14. *The throughput timeline for split live migration (w/ Cache) using eager or deferred mechanism.*

6.3 Eager Migration vs. Deferred Migration

Pragh provides two optional migration mechanisms, eager and deferred, to balance the accuracy and the timeliness of live migration. Fig. 14 compares these two mechanisms using the traversal benchmark. The monitor on each machine tracks remote accesses and reports the statistics to the coordinator periodically. After receiving statistics at 0 second, the coordinator adopts different mechanisms to notify migration threads. For eager migration, all of the migration threads will start migration directly, and the throughout reflects the benefits immediately, increasing from 239K to 2,142K. However, since the migration plan may not be optimal, the second migration happens at the next interval (after about 10s). The throughput further increases to 2,362K. For deferred migration, the coordinator will only ask monitors to track the local accesses on the potential key-value pairs for migration at 0 second, and do the migration with an optimal plan at the next interval. The throughput will directly increase from 239K to about 2,362K.

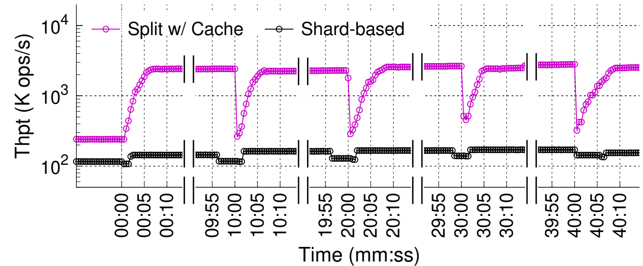


Fig. 15. *The throughput timeline for dynamic workloads using shard-based or split live migration.*

6.4 Dynamic Workloads

To study the effectiveness of split live migration in the face of dynamic workloads, we change workloads every 10 minutes by using non-overlapping scopes of starting vertices. As shown in Fig. 15, the performance notably drops every time the workloads change, because the location of vertex migrated for the current workload is very likely not suitable for the next workload. Shard-based migration can only provide very limited performance improvement as expected. Split migration with location cache can recover the performance after migration. Note that Pragh uses instant migration in this case, which is hard to implement in traditional migration approaches. When the monitor detects the frequency of accesses to some remote key-value pair exceeding a threshold (100 times per second), it will instantly report to the coordinator. Further, the migration on every machine can move values at any time, and there is no need to synchronize with other machines. Therefore, the performance is recovered gradually in about 5 seconds. Note that using a more aggressive policy could further reduce the time spent in recovery.

6.5 Application: RDF Graph and SPARQL Query

Wukong+M. To demonstrate the generality of Pragh, we have integrated *split live migration* with Wukong [52], called Wukong+M. Wukong is a state-of-the-art distributed graph store that leverages RDMA-based graph exploration to provide highly concurrent and low-latency SPARQL queries over large RDF graph datasets. RDF (Resource Description Framework) is a standard data model for the Semantic Web, recommended by W3C [2], which presents linked data as a set of $\langle subject, predicate, object\rangle$ triples forming a directed and labeled graph. SPARQL is the standard query language for RDF datasets, which can be supported by using graph exploration (i.e., graph traversals in parallel). We also implement an RDMA-friendly location cache on Wukong+M, similar to DrTM-KV [61].

Benchmark and workload. We use the Lehigh University Benchmark (LUBM) [3] which is widely used to evaluate the performance of RDF query systems [63, 36, 28, 52, 64, 37]. More specifically, we use LUBM-10240 dataset where each machine deploys about 32GB memory. We use the query set published in Atre et al. [8] and a mixed workload consist-

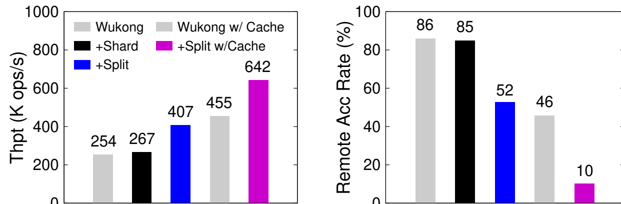


Fig. 16. *The comparison of (a) throughput and (b) remote access rate using a mixed workload for Wukong with various settings.*

ing of 6 classes as the same in the original paper [52]. The workload is skewed such that the starting vertices are chosen following a Zipf distribution ($\theta = 0.99$) over all vertices.

Performance. As shown in Fig. 16, Wukong+M (+Split) with location cache (w/ Cache) can outperform all other counterparts by up to 2.53×, thanks to split live migration that eliminates about 88% remote accesses (from 86% to 10%). Shard-based live migration (+Shard) only improves the mixed query throughput by about 5%, since it is hard to balance requirements for keys in each shard. The basic split migration (+Split) outperforms shard-based migration by 1.52× (407K ops/s vs. 267K ops/s) due to allowing fine-grained migration. After enabling location cache (+Split w/ Cache), the throughput further increases by 1.58× (642K ops/s vs. 407K ops/s).

7 Related Work

Live migration on relational stores. There have been many efforts to provide live migration features for distributed relational databases, considering different low-level architectures, such as shared-storage [20, 21, 26, 48, 19, 10] or partitioned database [53, 25, 60, 35]. They mainly focus on migrating shards efficiently across machines for balancing load and reducing latency. There are two main types of approaches: pre-copy based [20, 21, 60] and post-copy based [26, 25, 35].

To the best of our knowledge, almost all such systems adopt shard-based mechanisms (e.g., range or hash partitioning [17, 43]) and the changes of the ownership of shard are necessary when migration. Hence, they must maintain the state of shards explicitly by using internal global data structures or external location services [55, 4, 5]. Differently, split live migration fixes the (logical) location of data to avoid the maintenance overhead, which makes it different from all of the previous approaches.

The inherent drawback of one-off sharding has driven a few recent efforts to support dynamic sharding [25], auto sharding [4] and application-specific sharding [5] techniques. However, when shards still serve as the unit of migration, it is hard to balance the effectiveness (granularity) and efficiency (CPU and memory) for large-scale graph data with dynamic workloads due to lacks of locality.

Live migration on graph stores. The increasing importance of graph data models has stimulated a few recent designs of vertex migration or graph re-partitioning techniques targeting graph systems [44, 62, 34, 41, 65], since it is hard or even impossible to handle dynamic workloads or evolving graphs only relying on static partition-based approaches [27, 13]. The most related work is Mizan [34], a distributed graph processing system that leverages fine-grained vertex migration to improve *load balance* for iterative analytics workloads (e.g., PageRank and DMST [34]) over a *static* graph. Further, vertex migration can only happen when all worker threads reach a synchronization barrier (*stop-the-world*), and all selected vertices in one machine can only be migrated to a pairwise machine (*non-flexible*). By contrast, Pragh uses live migration to preserve *locality* for *concurrent* and *dynamic* traversal operations over *evolving* graphs. Thus, it makes many fine-grained migrations *on demand*, and vertices can be migrated to any machines *flexibly*.

Most graph re-partitioning approaches [44, 62, 65] need to maintain global metadata to map vertices to partitions, and use multiple phases to iteratively migrate vertices for reducing the communication cost. Therefore, these design choices make them slow to react to changes of workloads and other real-time events. Pragh can provide instant response to migration demands using lightweight monitoring and unilateral migration protocol.

Further, data replication has been used to improve the locality of traversal workloads over graph stores [29, 62, 40] by duplicating vertices on multiple machines. However, it will consume more memory and complicate the design of graph store in the face of evolving graphs. It should be noted that data replication is orthogonal to live migration, and integrating split live migration with fine-grained vertex replication [27, 13] is part of our future work.

8 Conclusion

This paper presents Pragh, an efficient locality-preserving live migration scheme for graph store. The key idea of Pragh is *split live migration*, which allows fine-grained migration while avoiding the need to maintain excessive metadata. Several key designs like the unilateral migration protocol, the integration of location-based caching, and the check-and-forward mechanism for evolving graphs made Pragh fast and full-fledged. Evaluations using both a graph traversal benchmark and SPARQL workloads confirmed the effectiveness and generality of Pragh.

Acknowledgments

We sincerely thank our shepherd Dushyanth Narayanan and the anonymous reviewers for their insightful suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 61772335, 61572314, 61732010), the National Youth Top-notch Talent Support Program of China, and a research grant from Alibaba Group through Alibaba Innovative Research (AIR) Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] IEEE 1588 Precision Time Protocol (PTP) Version 2. http://sourceforge.net/p/ptpd/wiki/Home/.

[2] Semantic Web. https://www.w3.org/standards/semanticweb/.

[3] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[4] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *OSDI*, pages 739–753, 2016.

[5] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the shards: managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 445–460, 2018.

[6] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. 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. ACM.

[8] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[9] D. Barak. VERBS Programming Tutorial. *OpenSHMEM*, 2014.

[10] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy. "cut me some slack": Latency-aware live migration for databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 432–443, New York, NY, USA, 2012. ACM.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[14] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[15] 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'10, pages 143–154. ACM, 2010.

[16] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[17] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[18] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, 6(11):1150–1161, Aug. 2013.

[19] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, Apr. 2013.

[20] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *CS, UCSB, Santa Barbara, CA, USA, Tech. Rep*, 9:2010, 2010.

[21] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[22] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414. USENIX Association, 2014.

[23] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[24] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *Proc. VLDB Endow.*, 9(11):852–863, July 2016.

[25] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[26] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[27] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[28] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, New York, NY, USA, 2014. ACM.

[29] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380, June 2016.

[30] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, pages 11–11. USENIX Association, 2010.

[31] B. Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 25–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM'14, pages 295–306. ACM, 2014.

[33] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 485–500, Santa Clara, CA, Mar. 2016. USENIX Association.

[34] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[35] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[36] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, 2013.

[37] C. Mayer, R. Mayer, J. Grunert, K. Rothermel, and M. A. Tariq. Q-graph: preserving query locality in multi-query graph processing. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, page 6. ACM, 2018.

[38] Mellnox. RDMA Aware Networks Programming User Manual, Rev 1.7. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[39] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114. USENIX Association, 2013.

[40] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM.

[41] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.

[42] A. Pavlo, C. Curino, and S. 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, New York, NY, USA, 2012. ACM.

[43] A. Pavlo, C. Curino, and S. 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, New York, NY, USA, 2012. ACM.

[44] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 375–386, New York, NY, USA, 2010. ACM.

[45] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, Aug. 2018.

[46] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret. Structural properties as proxy for semantic relevance in rdf graph sampling. In *International Semantic Web Conference*, pages 81–96. Springer, 2014.

[47] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, Dec. 2017.

[48] O. Schiller, N. Cipriani, and B. Mitschang. Prorea: Live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

[49] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[50] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, Nov. 2016.

[51] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[52] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proc. OSDI*, 2016.

[53] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[54] Titan. Titan Data Model. http://s3.thinkaurelius.com/docs/titan/current/data-model.html, 2018.

[55] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *USENIX Annual Technical Conference*, 2011.

[56] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[57] S. Wang, C. Lou, R. Chen, and H. Chen. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *Proc. USENIX ATC*, 2018.

[58] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[59] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[60] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 335–347, Santa Clara, CA, 2017. USENIX Association.

[61] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[62] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 517–528, New York, NY, USA, 2012. ACM.

[63] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276. VLDB Endowment, 2013.

[64] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proc. SOSP*, 2017.

[65] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 121–132. IEEE, 2016.

ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores

Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, Yinlong Xu
University of Science and Technology of China

Abstract

LSM-tree based key-value (KV) stores suffer from severe read amplification because searching a key requires to check multiple SSTables. To reduce extra I/Os, Bloom filters are usually deployed in KV stores to improve read performance. However, Bloom filters suffer from false positive, and simply enlarging the size of Bloom filters introduces large memory overhead, so it still causes extra I/Os in memory-constrained systems. In this paper, we observe that access skewness is very common among SSTables or even small-sized segments within each SSTable. To leverage this skewness feature, we develop ElasticBF, a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness. ElasticBF is orthogonal to the works optimizing the architecture of LSM-tree based KV stores, so it can be integrated to further speed up their read performance. We build ElasticBF atop of LevelDB, RocksDB, and PebblesDB, and our experimental results show that ElasticBF increases the read throughput of the above KV stores to 2.34×, 2.35×, and 2.58×, respectively, while keeps almost the same write and range query performance.

1 Introduction

With the exponential growth of data volume, traditional relational database meets challenges in scalability in dealing with extremely large-scale data. As an alternative, key-value (KV) store is widely used as the fundamental storage infrastructure in many applications, such as cloud systems [25], advertising [8, 16], social networks [3, 37], search indexing [8, 21], online gaming [13], etc. According to the used index structures, KV stores can be categorized into hash index based design [12,13,26], B-tree based design [20,31] and LSM-tree based design [27, 30, 32, 36]. Because hash index based design requires large memory and can not well support range query, and B-tree based design involves an abundance of random writes, so most modern KV stores use LSM-tree, e.g., LevelDB [21] at Google, RocksDB [16] at Facebook, Dynamo [14] at Amazon, and Cassandra [1] at Apache.

An LSM-tree based KV store is typically composed of two components, and we take LevelDB as an example to illustrate. One resides in memory to cache KV pairs, and it includes a MemTable and an Immutable MemTable. The other is stored in secondary storage, which is divided into multiple levels consisting of multiple SSTables. Each SSTable contains a set of sorted KV pairs and necessary metadata. When a level reaches its size limit, its SSTables will be compacted into the next level via compaction, which first reads out the SSTables in the two levels, then performs a merge sort, and finally writes back the new SSTables into the next level. As a result, compaction induces severe write amplification [27, 32].

Various designs are recently proposed to mitigate the compaction overhead [5, 27, 32, 42]. The most recent work PebblesDB [32] proposes a novel data structure called Fragmented Log-Structured Merge-Trees (FLSM), which follows the LSM-tree structure, but relaxes the fully sorted constraint in each level. Thus, compaction just needs to append data to multiple fragments in the next level, and there is no need to merge with the SSTables in the next level. Therefore, PebblesDB can greatly reduce the write amplification. However, sacrificing the sorted nature of KV pairs in each level, PebblesDB inevitably degrades the read performance.

On the other hand, LSM-tree based KV stores also suffer from severe read amplification [27, 31, 39]. This is because when lookup a KV pair, KV store needs to check multiple SSTables from the lowest level to the highest level until the key is found or all levels have been checked. Furthermore, it is required to read multiple metadata blocks to really check whether a KV pair exists in one SSTable. Thus, read amplification can reach a factor of over 300× [27]. To reduce extra I/Os induced by checking multiple SSTables, modern designs use Bloom filters in KV stores [35] to quickly check the existence of a KV pair in an individual SSTable. However, Bloom filters suffer from false positive, so they may return a positive result even when a KV pair does not exist in the SSTable, and this incurs unnecessary I/Os. We also conduct experiments to measure the impact of false positive by using Bloom filters with four `bits-per-key` in a 100GB KV store. Other parameter settings and system configurations are presented in §4.1. Results show that a key lookup incurs 7.6 inspections to SSTables on average, but 1.3 disk accesses are needless and induced by false positive.

Allocating more bits for each key can reduce the false positive rate, but the volume of all Bloom filters also increases, e.g., a 10TB KV store with 100B KV pairs requires 128GB Bloom filters under 10 `bits-per-key` setting. If the volume of Bloom filters exceeds the memory capacity, some filters will be swapped out to secondary storage, which induces extra I/Os and increases the read amplification.

It is a consensus that data access is usually skewed in real

applications [4, 9]. Some KV pairs are hot and frequently accessed, while most KV pairs are seldom accessed. Thus, if we allocate more bits to the Bloom filters for hot KV pairs, then we can reduce the overall false positive rate in the whole process of running an application, while still limit the volume of all Bloom filters. However, it is not an easy task to dynamically adjust the setting of Bloom filters in KV stores due to the following reasons, which are revealed by our experiments. First, different levels exhibit significantly different access unevenness. Even though KV pairs in a lower level are usually more frequently accessed, there are still a considerable proportion of SSTables in a higher level that are evidently hotter than SSTables in a lower level. Second, even for KV pairs within the same SSTable, the access unevenness is also serious. Last but not least, the hotness of KV pairs dynamically changes during the running time of applications. Monkey [10] proposes a heterogeneous scheme which allocates more bits to Bloom filters in lower levels, but it uses the same setting for all filters in the same level and fails to dynamically adjust the setting according to data hotness.

In this paper, we propose ElasticBF, a fine-grained and elastic Bloom filter management scheme. Its basic idea is assigning multiple small-sized Bloom filters to each small group of KV pairs when building SSTables, and these Bloom filters reside in secondary storage and are dynamically loaded into memory to activate according to the hotness of KV pairs. However, to realize dynamical allocation of Bloom filters, the following key issues must be addressed: (1) How to accurately measure and record the hotness of KV pairs with low storage and CPU overheads? (2) How to dynamically change the ability of Bloom filters based on hotness with low memory and computation overheads? (3) How to efficiently inherit the hotness of SSTables with low metadata overhead when SSTables are reorganized during compaction? ElasticBF carefully addresses the above issues by developing multiple techniques to limit its overhead, including *fine-grained allocation*, *hotness inheritance* and *in-memory management optimization*.

We emphasize that ElasticBF is orthogonal to existing works focusing on the optimization of KV store structures, so it can be combined with these designs to further accelerate the lookup of KV pairs. To demonstrate its efficiency, we carefully build ElasticBF atop three commonly used or state-of-the-art KV stores, LevelDB, RocksDB and PebblesDB. Experiments show that for all of the above KV stores, ElasticBF increases the read throughput to 2.34×, 2.35×, and 2.58×, respectively, while keeps almost the same write performance. Also, for workloads with mixed reads and writes, ElasticBF reduces read latency by 38.9% - 51.8% without affecting writes. Compared with Monkey, which is the state-of-the-art heterogeneous Bloom filter management scheme, ElasticBF achieves up to 2.20× throughput. In summary, our contributions are as follows.

- **Fine-grained allocation.** We find that the hotness of KV pairs varies significantly in different ranges within the same SSTable. So we divide each SSTable into different segments, measure and record their hotness with acceptable storage and CPU overheads, so as to receive relatively accurate hotness estimation and enable fine-grained Bloom filter allocation.

- **Hotness inheritance.** We design an effective scheme to estimate the access frequencies of new SSTables during compaction by inheriting from the outdated SSTables. So ElasticBF can preserve the hotness during the whole execution process of applications, and avoids frequent cold start of hotness due to compaction and consistently improves read performance.

- **In-memory management optimization.** We propose a Multi-Queue to manage the in-memory Bloom filters, and use parallel I/Os to accelerate the adjustment. Thus, we can dynamically adjust the Bloom filters in memory according to the hotness of KV pairs with negligible CPU overhead.

The rest of this paper is as follows. §2 introduces LSM-tree based KV stores and shows our observations on access skewness to motivate this work. §3 presents the design details of ElasticBF and §4 evaluates its performance. Finally, §5 reviews related work and §6 concludes this paper.

2 Background & Motivation

In this section, we first briefly introduce LSM-tree, then analyze the read amplification in LSM-tree based KV stores, and finally present our observations about data access locality in KV stores to motivate this work.

2.1 Log-Structured Merge Trees

Figure 1 illustrates the structure of an LSM-tree based KV store, which mainly consists of two components. One is in memory, which includes a *MemTable* and an *Immutable MemTable*. The other is in secondary storage, which is divided into multiple levels, say L_0, L_1, $\cdots$, L_k, where k depends on the KV store size. Besides, the size limit of level L_i is usually m times of that of level L_{i-1} for $1 \leq i \leq k$, e.g., $m = 10$ in LevelDB by default.

Now we illustrate how data is stored. Specifically, KV pairs are first written to the *MemTable* which works as a cache. When *MemTable* is filled up, it will be converted to an *Immutable Memtable*, which can not be written any more.

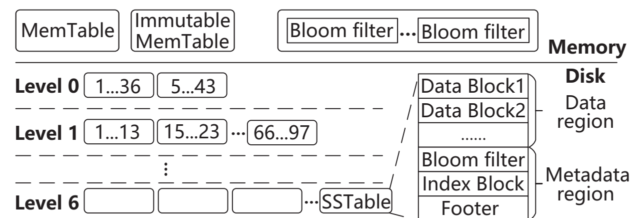


Figure 1: Structure of LSM-tree based KV store.

Later, *Immutable Memtable* will be packed into an SSTable and appended into L_0 in the secondary storage. Note that keys in each SSTable are sorted, but they are not sorted between SSTables in L_0 so as to make the writes to disk fast. Thus, this write policy degrades read performance, so the size of L_0 is usually limited, e.g., it is limited as 12 SSTables in LevelDB. To balance read and write performance, SSTables are organized into a multi-level tree, and if one level (say L_{i-1}) is filled up, its SSTables will be merged into its higher level (say L_i) by compaction, which merges all KV pairs in L_{i-1} into L_i, so data are sorted in every level except for L_0.

To find a key in the secondary storage, we need to search it level by level from L_0 to L_k. Note that we should check all SSTables in L_0 because they are not sorted, while we only need to check one SSTable in each of the other levels until we find the key or all levels are checked. Thus, we usually need to read multiple SSTables to find a key, which induces read amplification, and Bloom filters are commonly used to reduce the read amplification. As a result, besides KV pairs, each SSTable also includes Bloom filters and other metadata (see Figure 1). For performance consideration, the Bloom filters are usually required to be also buffered in memory.

However, Bloom filters suffer from false positive because of hash collision, and thus incur extra I/Os to read out data from SSTables for key comparison. The false positive rate of a Bloom filter is $(1-e^{-k/b})^k$, where b is the number of bits allocated to each key, i.e., `bits-per-key`, and k means the number of hash functions [24]. Since $(1-e^{-k/b})^k$ is minimized when $k = ln2 \cdot b$, false positive rate can be simply represented as 0.6185^b. Thus, the value of b directly determines the memory usage of a Bloom filter. We can reduce the false positive rate by allocating more `bits-per-key` for Bloom filters, but allocating more bits to each keys will increase the volume of all Bloom filters and thus consumes more memory. Even worse, if the volume of all Bloom filters exceeds the memory capacity, some Bloom filters will be swapped out to secondary storage, and this will induce extra I/Os and further aggravate read amplification.

2.2 Motivation

Uneven accesses are still very common in KV stores [9, 22], where only a small proportion of the KV pairs are frequently accessed, while the majority of the KV pairs are seldom accessed. Therefore, if we allocate more bits to the Bloom filters for hot KV pairs and fewer bits for cold ones, then the overall positive false rate during the whole execution process of applications will be reduced. Clearly, we will face to a series of challenges to realize such heterogeneous Bloom filters and enable dynamic adjustment. In this subsection, we present our observations on the access skewness of KV stores to motivate this work. The detailed design of ElasticBF will be presented in §3.

We run experiments with RocksDB to validate the access unevenness in KV stores. We use YCSB [9] to load a 256GB

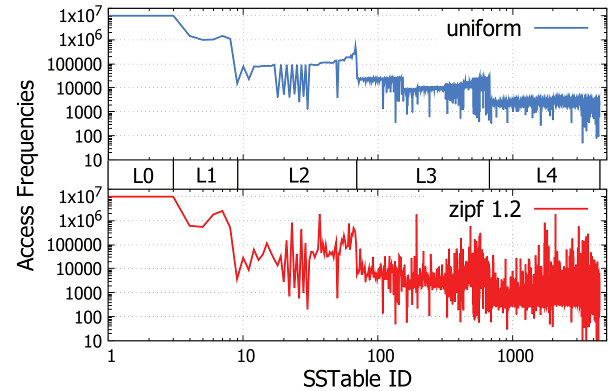


Figure 2: File access frequencies under different workloads.

database in the experiments, where the size of each SSTable is set as 64MB, which is the default configuration, and the size of each key value pair is 1KB. Note that the maximum size of L_1 is configured as 256MB in RocksDB, and the size of L_i is 10 times of that in L_{i-1} ($i \geq 2$), therefore 5 levels are enough to keep 256GB data. We then generate two representative workloads with uniform and Zipfian distributions, and each workload contains ten million Get requests. Note that there are about 4400 SSTables in the tested KV store, so issuing ten million Get requests is enough to study the access pattern. To make the evaluation of the hotness of SSTables and the hotness of different regions in the same SSTable accurate, we disable the Bloom filters in these experiments. That is, we search the keys level by level, and at each level, we compare the target key with the key ranges of SSTables. If the key falls into the range of an SSTable, we will read the data out and check whether the key exists or not until the key is found or all levels are checked.

We first show the file-level access characteristics, and Figure 2 shows the access frequency of each SSTable. The x-axis represents the identities of SSTables which are numbered sequentially from the lowest level to the highest level, and the y-axis shows the number of accesses to each SSTable. From the results, we can have two observations. First, on average, the access frequencies of SSTables in lower levels are higher than those in higher levels. This is because lookup always flows from lower levels to higher levels. Second, if we zoom in one particular level, we can find that the access frequencies vary very significantly from SSTables, i.e., some SSTables are much hotter than others within each level. Besides, when we compare the access frequencies of SSTables in adjacent two levels, we can find that it is very common to have some SSTables in level L_{i+1} which are even hotter than some SSTables in level L_i, especially for the skewed workload with Zipf distribution. That is, SSTables in higher levels may also be hotter than those in lower levels. For example, 21% of SSTables in L_4 is even hotter than 11% of SSTables in L_3. More importantly, since more than 98% SSTables are stored in the highest two levels, i.e., L_3 and L_4 in this example, we can conclude that the hotness of most SSTables can not be

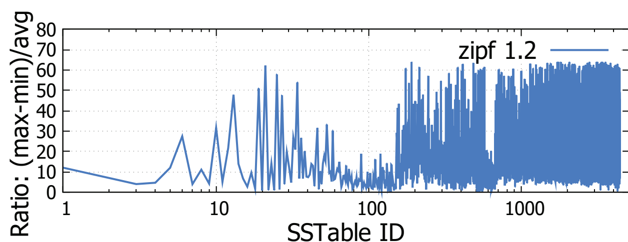


Figure 3: The ratio of the difference between the maximum and the minimum access frequencies of different regions to their average access frequency within each file.

accurately characterized according to which level they are placed. This observation implies that Monkey's level-based coarse-grained heterogeneous Bloom filters can not take full advantage of the access skewness, and finer-grained Bloom filter design is necessary.

Since the size of an SSTable may still be configured to be large to leverage sequential I/O bandwidth, e.g., RocksDB uses 64MB or even larger SSTables, the access skewness may still be serious within each SSTable. Note that it will bring very large memory and CPU overheads to record the hotness of each KV pair, so we divide each 64MB SSTable into 64 regions, each of which has 1MB, and record the access frequencies of different regions in each SSTable. Figure 3 shows the ratio of the difference between the maximum and the minimum access frequencies of all regions in the same SSTable to their average access frequency. We can see the ratio value is very large for many SSTabes, e.g., greater than 10 for 73% SSTables. So the access unevenness is very serious even within the same SSTable.

In summary, access skewness is very serious among different SSTables, and even among different regions within the same SSTable. This offers an opportunity to develop finer-grained heterogeneous Bloom filters by allocating more `bits-per-key` for hot SSTables or regions so as to reduce the overall false positive rate without increasing the volume and memory overhead of Bloom filters.

3 Design

The main idea of ElasticBF is to construct multiple Bloom filters for each SSTable, but allocate less `bits-per-key` to each filter. Note that the Bloom filters in SSTables are stored in secondary storage, and they are just reserved for future use. That is, Bloom filters become active only after being loaded into memory, which is a dynamical process according to the hotness of SSTables. If an SSTable becomes hot, we will load more of its Bloom filters into memory, and we may also disable some of its Bloom filters in memory when it becomes cold. Thus, we can dynamically adjust Bloom filters while avoiding heavy I/O and CPU overheads for computing the hash functions when we change Bloom filters according to the hotness of SSTables. Thanks to the dynamic allocation and adjustment of Bloom filters, we can reduce the overall false positive rate, while keeping the same memory usage.

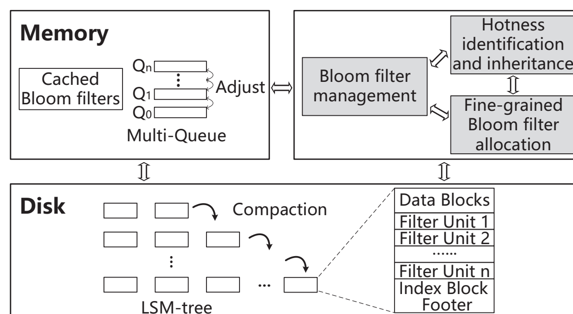


Figure 4: The architecture of ElasticBF.

3.1 Overview

Figure 4 depicts the architecture of ElasticBF, which mainly contains three components, *fine-grained Bloom filter allocation*, *hotness identification and inheritance*, and *Bloom filter management in memory*. For the design of fine-grained Bloom filter allocation, we mainly face to the problems of how many Bloom filters should be allocated to each SSTable and how many bits should be assigned to each filter so as to achieve low false positive rate and low memory usage. We also need to carefully design the data structure and management scheme with low I/O overhead. For hotness identification, our goal is to achieve relatively accurate estimation of hotness with low overhead. Finally, hotness inheritance is designed to avoid cold start of hotness identification after compaction, and Bloom filter management in memory is to efficiently adjust Bloom filters according to hotness.

Remarks: ElasticBF realizes an elastic Bloom filter management scheme with little extra memory usage and small CPU and I/O overheads. It is orthogonal to existing works focusing on optimizing the structure of KV stores and can be integrated to accelerate their read performance. Besides, ElasticBF may also be applied in other scenarios to improve object lookups, and the management technique for hot/cold adaption is applicable to other summary data structures.

3.2 Fine-grained Bloom Filter Allocation

The read performance of KV stores can be improved by reducing the I/Os caused due to false positive of Bloom filters. In the following, we first analyze the expected false positive rate by dynamically activating Bloom filters according to hotness, then we describe how to construct multiple Bloom filters for SSTables to realize fine-grained allocation.

Construction of multiple Bloom filters. ElasticBF generates multiple Bloom filters for each SSTable by using different and independent hash functions. Each filter is allocated with less `bits-per-key`, and we call it a *filter unit*. All filter units assigned to an SSTable are named as a *filter group*, as shown in Figure 5. Since the multiple filters within a filter group are independent, a key is certainly not in an SSTable as long as one filter unit returns a negative answer. That is, if multiple filter units are enabled, then only when all enabled

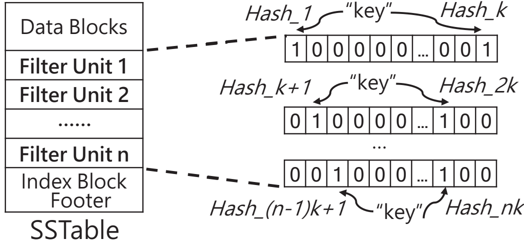


Figure 5: Construction of multiple Bloom filters.

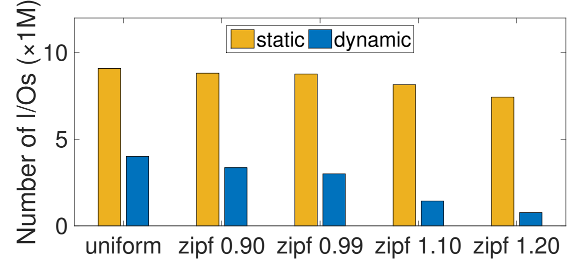


Figure 6: Number of I/Os caused due to false positive.

filter units indicate the existence of a key, we need to read out the SSTable to search the key.

As pointed out in [24], the false positive rate of a filter group is equivalent to a single Bloom filter which has the same `bits-per-key` to all filter units within the filter group. We call this feature *separability*, which can be further justified as follows. Assume that a filter group consists of *n* filter units, each of which is a b/n `bits-per-key` filter, then the false positive rate of each filter unit is $0.6185^{b/n}$. Since the filter units in a group are generated by different independent hash functions, the false positive rate of n filter units in a group is $(0.6185^{b/n})^n = 0.6185^b$, which is exactly the same with that of a single Bloom filter with b `bits-per-key`.

Based on the separability feature, we should determine b and n to optimize the setting of multiple Bloom filters. As we should enable all the filter units in a group for the hottest SSTables such that the false positive rate 0.6185^b closes to zero, in our configuration, we set $b = 24$ with a false positive rate of about 0.001%. On the other hand, n indicates the maximum number of hotness categories to distinguish different SSTables. Increasing n will more accurately differentiate the hotness of SSTables, but it needs more I/Os to load filter units to achieve low false positive rate. Thus, we set $n = 6$, i.e., a filter group has 6 filter units, each is allocated with 4 `bits-per-key`. We will analyze the impact of `bits-per-key` of each filter unit on the read performance in §4. Notice that ElasticBF allocates multiple filter units to SSTables, which induces extra storage usage. Assuming that the size of KV pairs is 1KB, thus one group of filter units cost about 192KB storage, which is only 0.3% of a 64MB SSTable, and the filters are stored in secondary storage, so the storage overhead of ElasticBF is negligible.

Benefit analysis. Now we analyze the benefit of using multiple Bloom filters. Suppose that there are N SSTables, $s_1, s_2, ..., s_N$, in a KV store, and we use static setting to set b `bits-per-key` for all SSTables, i.e., the memory usage to reside Bloom filters is b bits for each key. Suppose that a workload needs to access SSTable s_i with p_i times, then the expected number of times to really read out data from all SSTables due to false positive with static setting is

$$R_{static} = \sum_{i=1}^{N} p_i \cdot 0.6185^b. \quad (1)$$

In a contrary, if we dynamically set multiple Bloom filters with the same memory usage as the static setting, and suppose we load n'_i filter units for SSTable s_i according to its hotness, each allocates b' `bits-per-key`, then under the assumption of the same memory usage with static setting, the expected number of times to read out data from SSTables is

$$R_{dynamic} = \sum_{i=1}^{N} p_i \cdot \left(0.6185^{b'}\right)^{n'_i}, \quad (2)$$
$$\text{subject to} \quad \sum_{i=1}^{N} n'_i \times b' \le Nb,$$

where the inequality $\sum_{i=1}^{N} n'_i \times b' \le Nb$ represents the same memory usage constraint.

To better understand why dynamical allocation can reduce I/Os, i.e., $R_{dynamic} < R_{static}$, we count the number of I/Os due to false positive by conducting experiments on RocksDB. We set the average `bits-per-key` as 4 for both Bloom filter allocation schemes. For the dynamical allocation scheme, we generate 6 filter units for each SSTable and still use 4 `bits-per-key` for each filter unit. We first issue ten million Get requests on a 100GB database using static setting of Bloom filters, half of the Get operations request non-existent items, and we count the real number of I/Os issued due to false positive as R_{static}. Then we classify SSTables into 7 categories (C_0, C_1, ..., C_6) according to their access frequencies, and initially load i filter units for SSTables in C_i in the dynamical allocation. We use this configuration to replay the ten million Get requests and count $R_{dynamic}$. Figure 6 shows the results under different workloads, we find that with the same memory usage, dynamic setting of Bloom filters reduces the number of I/Os caused due to false positive under different workloads by 55.9% - 89.7% compared to static setting, and the reduction becomes larger for more skewed workloads. Note that in practical systems, Bloom filters may be configured with larger `bits-per-key` so as to achieve very low false positive rate, dynamical allocation still has its benefit, e.g., it can use much less memory to achieve similar false positive rate.

Finer-grained design with chunking. As mentioned in Section 2.2, access unevenness is still serious within an SSTable. So we may further reduce the false positive rate by differentiating the hotness of keys within the same SSTable. However, this will bring too large extra overheads of memory usage and CPU for recording the hotness of individual keys. To balance the accuracy of measuring hotness and the extra

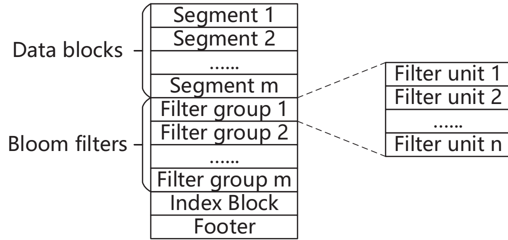


Figure 7: Segments in one SSTable.

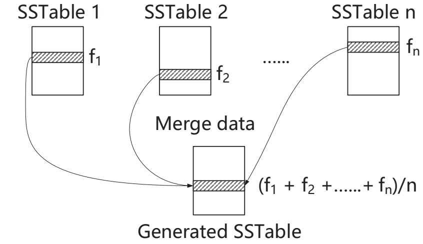


Figure 8: Hotness inheritance after compaction.

overheads to KV stores, we further divide each SSTable into multiple regions called *segments* and record the hotness at the granularity of the segment. Each segment is then allocated a group of filter units, as shown in Figure 7. From §2.2, we know the hotness of different segments still significantly varies, so we are still expected to reduce the false positive rate by differentiating the hotness of segments.

The challenging issue is to optimize the size of segments, as large segment can not accurately reflect the hotness of different KV pairs in an SSTable and small segment will bring large overhead to KV stores. Our rule to configure the size of segment is to make the size of each Bloom filter be close to the device block size, e.g., 4KB, so as to reduce the I/O overhead when loading Bloom filters. We will analyze the impact of segment size on the read performance in §4.

Finally, since we only need several bytes to record the hotness for each segment, the memory overhead is smaller than 1% of the Bloom filter size. For storage overhead, if the KV pair size is 1KB and ElasticBF uses 4 `bits-per-key` for each filter unit, then one filter unit only costs around 2KB storage space, which is only 0.05% of a 4MB segment.

Remarks: All Bloom filters allocated to an SSTable are stored in its metadata area and kept in secondary storage. Upon reading an SSTable, the default number of Bloom filters are also loaded into memory, so loading Bloom filters at initialization does not induce extra I/Os.

3.3 Hotness Identification and Inheritance

Hotness identification. The hotness of a segment is determined by its access frequency and the time duration since its last access. Specifically, we propose an *expiring policy* to differentiate hot/cold segments. We maintain a global variable named `currentTime`, which is defined as the total number of Get requests issued to the whole KV store so far, and we also associate a variable named `expiredTime` with each segment to denote the time point at which the segment will be "expired". Precisely, `expiredTime` is defined as `lastAccessedTime + lifeTime`, where `lastAccessedTime` denotes the time of the most recent access to the segment and `lifeTime` is a fixed constant. Note that the "time" concept here means logical time which is actually represented by the number of accesses. Each time when a segment is accessed, we increase the `currentTime` by one and update the `expiredTime` of this segment by setting `lastAccessedTime` as the updated value of the `currentTime`. We define a segment as "expired" if `currentTime` already becomes larger than `expiredTime`. The physical meaning of the above policy is that if a segment is not accessed during a fixed number of Get requests which is defined by `lifeTime`, then it is expired and considered as cold. Note that the time complexity to update the hotness metadata of a segment is only $O(1)$. Besides, the memory overhead is also small, e.g., for a 100GB KV store, there are around 25K segments whose size is 4MB, assume that 4 bytes are used to record the `expireTime` of each segment, then the total memory overhead is only around 100KB.

Hotness inheritance after compaction. Compaction will trigger merge sort between SSTables to generate new SSTables. So the segments in new SSTables are also newly generated and their hotness should be changed. If we set the hotness of new segments as 0, then ElasticBF will face to cold start of hotness and this may degrade the performance of future read from the new segments. To inherit the hotness, ideally, we can accurately estimate the hotness of new segments based on the hotness of all keys within it, but this will bring into KV stores too large overhead.

Instead, ElasticBF uses the hotness of old segments to estimate the hotness of new segments. Specifically, as illustrated in Figure 8, when a new segment is generated, we first find out the old segments which are involved in the procedure of generating the new segment and also have overlapped key ranges, then we estimate the hotness value of the new segment by simply using the mean of the hotness of all old segments. At last, we enable some filter units for the new segment accordingly. Experiments in §4 show that this simple scheme is efficient to improve the read performance for workloads with mixed reads and writes in KV stores.

3.4 Bloom Filter Management in Memory

Now the final issue is to determine how many filter units should be enabled for each segment. Although we can address this issue by formulating an optimization problem to minimize the overall false positive rate, but this will consume lots of CPU resources. Besides, every access changes the access frequency of some segment, so it needs to recompute the optimal solution and incurs lots of I/Os to adjust the optimal configuration. To address this issue, ElasticBF develops a lightweight and efficient adjustment scheme.

Bloom filter adjusting rule. We use a metric which is defined as the expected number of I/Os caused by false positive to guide the adjustment, and we denote this amount of extra I/Os as $E[\text{Extra_IO}]$, which can be expressed as

$$E[\text{Extra_IO}] = \sum_{i=1}^{M} f_i \times r_i, \quad (3)$$

where M means the total number of segments in the KV store, f_i denotes the access frequency of segment i, r_i denotes the false positive rate and it is determined by the number of filter units loaded in memory for segment i. Here, the rule of thumb is to adjust the number of filter units, and thus changes r_i, so as to make $E[\text{Extra_IO}]$ be decreased.

The procedure of adjusting Bloom filters is as follows. Each time when a segment is accessed, we update its access frequency and the $E[\text{Extra_IO}]$, then we check whether $E[\text{Extra_IO}]$ could be decreased if we enable one more filter unit for this segment and disable one unit for other segment to guarantee the same memory usage. If the $E[\text{Extra_IO}]$ could be decreased, then we apply the adjustment, otherwise, we do nothing. Note that in this adjusting procedure, one key issue is to find out which filter unit should be disabled, and we address this problem by maintaining an in-memory index based on Multi-Queue, which will be described later.

Realizing dynamic adjustment with Multi-Queue. Recall that the challenging issue in the adjust procedure is to decide which filter unit should be disabled. We extend Multi-Queue (MQ) [33,46] to address this problem. Specifically, ElasticBF maintains multiple in-memory Least-Recently-Used (LRU) queues to manage the metadata of each segment as shown in Figure 9. We denote these queues as Q_0,...,Q_n, where n is equal to the maximum number of filter units allocated to each segment. Each element of a queue corresponds to one segment, and it manages the filter units enabled for the segment. Precisely, each element in queue Q_i indicates that i filter units are enabled for the corresponding segment, i.e., only these i filter units are used to check the existence of keys. To keep the LRU feature of each queue, each time when a segment is accessed, we move the corresponding element to the MRU side within the same queue.

To find out which filter unit should be disabled and then removed from memory, we use the hotness information defined by the *expiring policy* described in §3.3. Specifically, we search "expired" segments from Q_n to Q_1, and for each queue, we search from the LRU side to the MRU side, since an "expired" segment must be the least recently used one. When we find an "expired" segment, and if the $E[\text{Extra_IO}]$ can be decreased when we disable one filter unit of this segment, we then downgrade it to the next lower-level queue to release one filter unit. Note that the access frequency of the "expired" segment does not change, while the $E[\text{Extra_IO}]$ could be decreased because of the change of false positive rates by adjusting the Bloom filters in corresponding segments. If there is no "expired" segment,

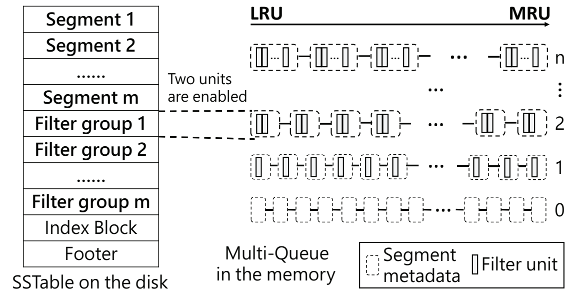


Figure 9: The in-memory Multi-Queue in ElasticBF.

we skip the Bloom filter adjustment this time, this is a conservative strategy to prevent us from degrading the lookup efficiency of possible hot segments (which are not "expired"), and combined with checking if $E[\text{Extra_IO}]$ (which is related to the access frequency) can be decreased, the adjustment overhead is limited as the adjustment frequency is limited, we also analyze the adjusting overhead in §4. On the other hand, we set `lifeTime` as the same order of magnitude as the total number of segments. The rationale is that if there is no "expired" segment, it means almost all the segments have been accessed recently, so they may have similar hotness during that time and we do not need to do the adjustment.

3.5 Implementation Issues

We implement ElasticBF on top of various commonly used KV stores, including LevelDB, RocksDB and PebblesDB. Here, we briefly describe the issues in the implementation.

ElasticBF keeps multiple filter units for each segment in each SSTable, to make minimum changes to SSTables, each filter unit is treated as a *meta block* in original SSTable organization, and the offset information in the file are recorded in *meta index block*. Besides, as generating multiple Bloom filters may add latency to writes, ElasticBF leverages multi-threading via *threadpool* to generate multiple filter units simultaneously so as to further reduce the computation time. On the other hand, ElasticBF maintains a background thread to manage Multi-Queue, so loading filter units and fetching data from secondary storage can be done in parallel, therefore the device bandwidth can be efficiently exploited.

4 Evaluation

In this section, we evaluate ElasticBF to validate its efficiency. We build ElasticBF atop LevelDB [21], RocksDB [16], and PebblesDB [32], and compare the performance of these systems with and without ElasticBF so as to study how much improvement ElasticBF can achieve. We point out that LevelDB is the classical LSM-tree based design, and RocksDB further improves the performance of LevelDB with multiple optimizations. Both of them are widely used as baselines for performance comparison [27, 32]. Besides, PebblesDB is developed based on the new and state-of-the-art index called fragmented LSM-tree, so we also take it as a baseline to demonstrate the effectiveness of ElasticBF. We

emphasize that since ElasticBF is orthogonal to the works optimizing architecture of KV stores, it can also be integrated to other KV stores to further speedup their read performance. In the evaluation, we try to address the following questions.

- How much improvement does ElasticBF achieve to speedup the read performance of KV stores? (§4.2)
- How is the performance of ElasticBF under the workloads of YCSB benchmark? (§4.3)
- What is the performance impact of dynamically allocating bits to Bloom filters in ElasticBF, as compared to the static heterogeneous scheme in Monkey? (§4.4)
- What is the performance impact of different configurations on ElasticBF? (§4.5)

4.1 Experiment Setup

We run experiments on a Dell PowerEdge R730 with an 12-cores Intel Xeon CPU E5-2650 v4 with 2.20GHz processor, 64GB RAM, and Ubuntu 16.04 OS with Linux 4.15 kernel. The testbed is equipped with one 500GB SSD and one 1TB 7200RPM HDD. By default, we run experiments on the SSD. We build ElasticBF on top of LevelDB (v1.20), RocksDB (v5.14) and PebblesDB. As RocksDB and PebblesDB use 64MB or larger SSTables as their default configuration, we also set the SSTable size by modifying *max_file_size* to 64MB in LevelDB. To make a fair comparison, only the management strategy of Bloom filters was changed accordingly, while the memory usage is limited as the same and other parameters are also set as the same with the default values.

In the experiment, we use the benchmark YCSB-C [31,34], which is the C++ version of YCSB [9] with low overhead. Unless specifically mentioned, we use the following default configuration. We set the size of each KV pair as 1KB, and load a 100GB database with randomly generated distinct keys. For the benchmarked workload, we generate 10M Get operations by following the Zipf distribution with a Zipfian constant 0.99 by default. Note that there is no warm up phase, i.e., we immediately issue the benchmarked workload to the randomly loaded database. We also point out that the performance is already stable after issuing 10M operations. By default, we assume that half of the Get operations request non-existent items (i.e., zero lookup), mainly because lookups of non-existent KV pairs are very common in practical systems [6, 23, 35, 38]. As many KV stores provide their own cache mechanisms, thus we enable *direct I/O* [19] to better manage memory. For the default setting, we disable the *block cache* [17] to minimize the influence of cache. This represents the scenario in which a KV store runs within a memory-constrained environment [11,25], we also show the performance impact of block cache size in §4.5. For ElasticBF, the segment size of each SSTable is set as 4MB, and the `lifeTime` is set as 10K as there are around 30K segments in total. The average Bloom filter space for each key (i.e., `bits-per-key`) is set as four bits, this is because allocating a large number of bits for each key is not cost-efficient and it may be impractical in very large KV stores. Cassandra [2] also uses a similar setting of about 5 `bits-per-key` by configuring the Bloom filter to have the false positive rate of 0.1 in its *LeveledCompactionStrategy*. We also study the impact of different system configurations on the performance of ElasticBF in §4.5.

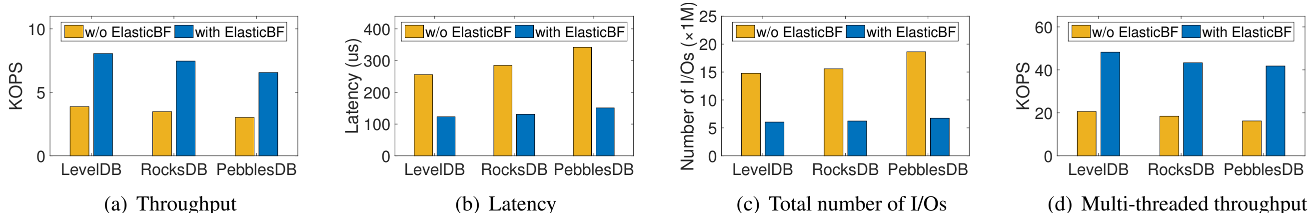
4.2 Micro-benchmarks

We first evaluate the performance of ElasticBF with micro-benchmarks. To evaluate the read performance, we consider both read-only workload and mixed workloads with different read/write ratios so as to validate the effectiveness of the hotness inheritance technique in ElasticBF. Finally, we also show the performance impact on writes and range queries.

Read-only workload. We use one thread to run the YCSB benchmark to perform 10M Get requests. Figure 10(a)-(c) show the results of read throughput, average read latency, and total number of I/Os. We can see that ElasticBF improves the read performance of different KV stores. Specifically, the read throughput with ElasticBF is increased to 2.08×, 2.15× and 2.17× compared to the results without ElasticBF under LevelDB, RocksDB and PebblesDB, respectively. For average read latency, ElasticBF can reduce the latency of LevelDB, RocksDB and PebblesDB by 51.9%, 54.0% and 55.8%, respectively. The improvement of ElasticBF is mainly because the reduction of extra I/Os caused by false positive of Bloom filters. To validate this, we also count the total number of I/Os generated to serve the Get requests, and the results are shown in Figure 10(c). We can see ElasticBF reduces the number of I/Os issued under different KV stores by 59.1% - 63.8%. As a result, ElasticBF can further improve the read performance of KV stores. To validate the effectiveness of the *expiring policy* and the adjusting rule, we count the total number of I/Os issued by loading filter units, it is only about 1% of the total number of I/Os generated to serve the Get requests, thus the adjusting overhead is small.

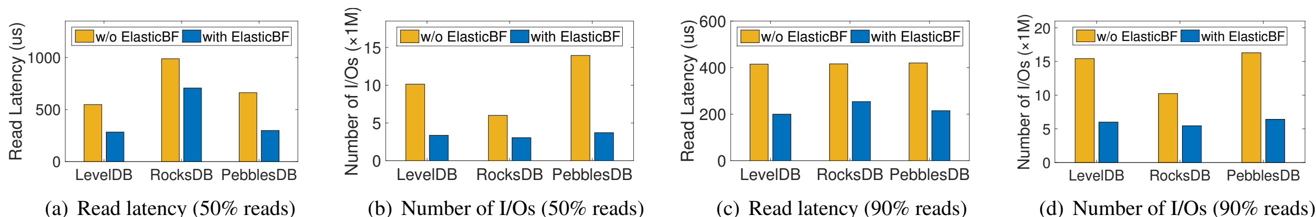
We further study the concurrent read performance of ElasticBF by using 16 threads to run the YCSB benchmark, and each thread performs 1M Get requests. Since we observe similar results for throughput, latency and total number of I/Os, we only show the throughput results in Figure 10(d) for the interest of space. In particular, ElasticBF increases the read throughput to 2.34× - 2.58× in these KV stores. Note that the improvement is slightly larger than that in single-threaded scenario, this is because multi-threaded reads can better utilize the I/O bandwidth.

Mixed workloads. Now we show the performance of ElasticBF under mixed workloads with different read/write ratios. The goal of this experiment is to validate that ElasticBF can still achieve a consistent improvement for reads due to the hotness inheritance design, even though compaction continuously generates new SSTables.



(a) Throughput (b) Latency (c) Total number of I/Os (d) Multi-threaded throughput

Figure 10: Read performance of KV stores with and without ElasticBF under read-only workload.

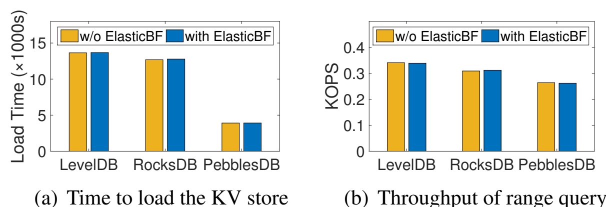


(a) Read latency (50% reads) (b) Number of I/Os (50% reads) (c) Read latency (90% reads) (d) Number of I/Os (90% reads)

Figure 11: Read performance of KV stores with and without ElasticBF under mixed workloads.

Figure 11 (a)-(b) show the results under the workload with 50% reads and 50% writes, and Figure 11 (c)-(d) show the results under the workload with 90% reads and 10% writes. Note that the total number of requests in the workload is 10M. We can see that ElasticBF can help reduce the read latency by 48.2%, 28.4%, and 54.8% for LevelDB, RocksDB and PebblesDB, respectively, under the workload with 50% reads, and the corresponding reduction ratios are 51.8%, 38.9% and 48.8% under the workload with 90% reads. We also count the total number of I/Os issued by Get operations. Specifically, ElasticBF reduces 66.8% (61.1%), 49.1% (46.7%), and 73.3% (60.7%), for LevelDB, RocksDB and PebblesDB, respectively, for workloads with 50% (90%) reads. Note that the reduction of read latency is smaller than that under read-only workload, the reason is that different KV stores use different compaction strategies, e.g., RocksDB enables multiple threads to do compaction and PebblesDB reduces compaction I/Os by avoiding rewriting SSTables to the same level, so the background compaction I/Os that are competed with the foreground Get I/Os are varied from KV stores. Note that the write performance does not decrease, and we will evaluate the write performance later.

Write and range query performance. Now we study the impact on write and range query performance. For different KV stores, we first randomly load a 100GB database and then issue 10M scan requests. We compare the time of loading the database and performing scan requests to evaluate the write and range query performance, and the results are shown in Figure 12. We can see that both the write and range query performance keep almost the same (the difference is less than 1%) even when ElasticBF is integrated in KV stores. The main reason is that Bloom filters are organized into blocks in SSTables, and ElasticBF also uses multi-threading to speedup the generation of Bloom filters. For range query, since it needs to fetch all blocks overlapped with the given range, Bloom filters are not involved in this procedure. Thus, ElasticBF has negligible impact on write and range query performance.



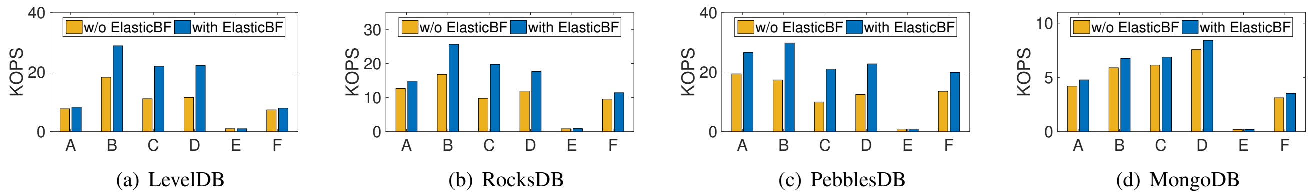
(a) Time to load the KV store (b) Throughput of range query

Figure 12: Put and Scan performance.

4.3 YCSB Benchmarks

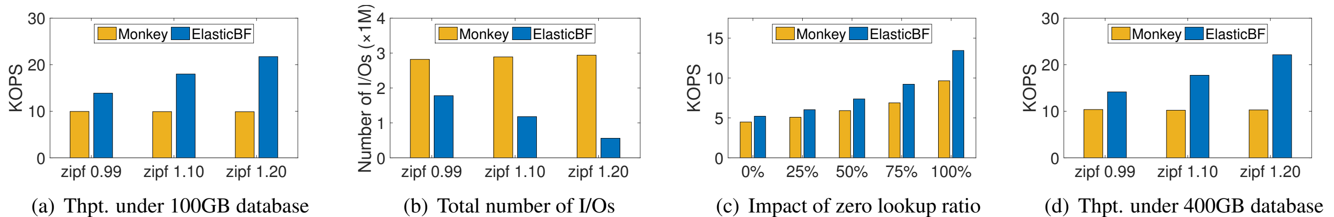
Now we evaluate the performance of ElasticBF with YCSB benchmarks, which provide a set of six workloads with different combinations of KV operations. Specifically, Workload A consists of 50% reads and 50% updates, Workload B consists of 95% reads and 5% updates, Workload C consists of 100% reads, Workload D consists of 95% reads and 5% inserts, Workload E consists of 95% scans and 5% inserts, and Workload F consists of 50% reads and 50% read-modify-writes. Note that Workload D uses the Latest distribution [9], while others follow Zipfian distribution. Each of the six workloads consists of 10M operations, which are issued on a 100GB database, and other settings are the same as before.

We first compare the performance of LevelDB, RocksDB and PebblesDB with and without ElasticBF. Figure 13(a), 13(b) and 13(c) show the throughput results. We can see that ElasticBF improves the performance for all workloads except for Workload E (95% scans), this is because Workload E is a scan-dominated workload, and ElasticBF does not affect the performance of write and scan. In particular, for read-only Workload C, ElasticBF achieves 1.99× - 2.11× throughput in different KV stores due to the optimized Bloom filter design. For Workload A (50% reads) and Workload B (95% reads), ElasticBF improves the throughput by 7.4% - 36.8% and 52.6% - 71.5% for different KV stores, respectively. The reason why the improvement under Workload A and B is smaller than that under Workload C is because the request keys are Zipfian distributed, and the updates make most of



(a) LevelDB (b) RocksDB (c) PebblesDB (d) MongoDB

Figure 13: Performance comparison of different KV stores with and w/o ElasticBF under YCSB benchmarks.



(a) Thpt. under 100GB database (b) Total number of I/Os (c) Impact of zero lookup ratio (d) Thpt. under 400GB database

Figure 14: Performance compared with Monkey under micro-benchmarks.

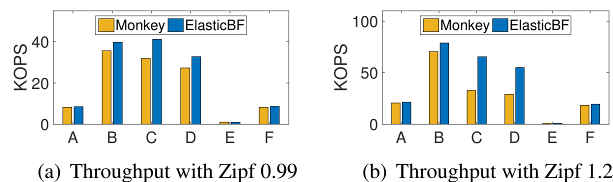
the accessed keys issued by Get are stored in lower levels, and thus leads to a smaller number of SSTables that need to be checked during read than that in read-only Workload C, so ElasticBF has a smaller improvement. The above reason also leads to the results under Workload F which has 50% reads and 50% read-modify-writes, and ElasticBF improves the throughput by 7.8% - 46.6%. Finally, for Workload D, ElasticBF increases the throughput by 47.9% - 93.0%.

We also study the performance impact of ElasticBF on MongoDB [29], which is a popular open-source NoSQL database using WiredTiger [41] and RocksDB as its storage engine. Since WiredTiger is not based on LSM-tree, we choose RocksDB as the storage engine, and evaluate the performance improvement when integrate ElasticBF in MongoDB. The client of YCSB benchmark is running on the same machine with the MongoDB server. Figure 13(d) shows the results. We find that the improvements are only about 11% - 15% except for the scan-dominated Workload E. This is because the YCSB workloads issue only one read/update/insert per request, while MongoDB adds a lot of latency in the critical path of read operations, e.g., the query planner, and thus the latency induced by RocksDB accounts for only about 20% of the total latency. As a result, optimization in the KV storage layer does not result in a large improvement, and this is also observed in PebblesDB [32]. However, we point out that MongoDB may issue batch reads to hide extra latency in real-world scenarios, and in this case, ElasticBF must bring in larger improvement.

4.4 Comparison with Monkey

Now we compare the performance of ElasticBF with Monkey [10], both are built atop LevelDB. As Monkey mainly focuses on zero lookups, which are very common in practice, e.g., the insert-if-not-exist queries, we also assume that all Get operations request non-existent items in the experiments.

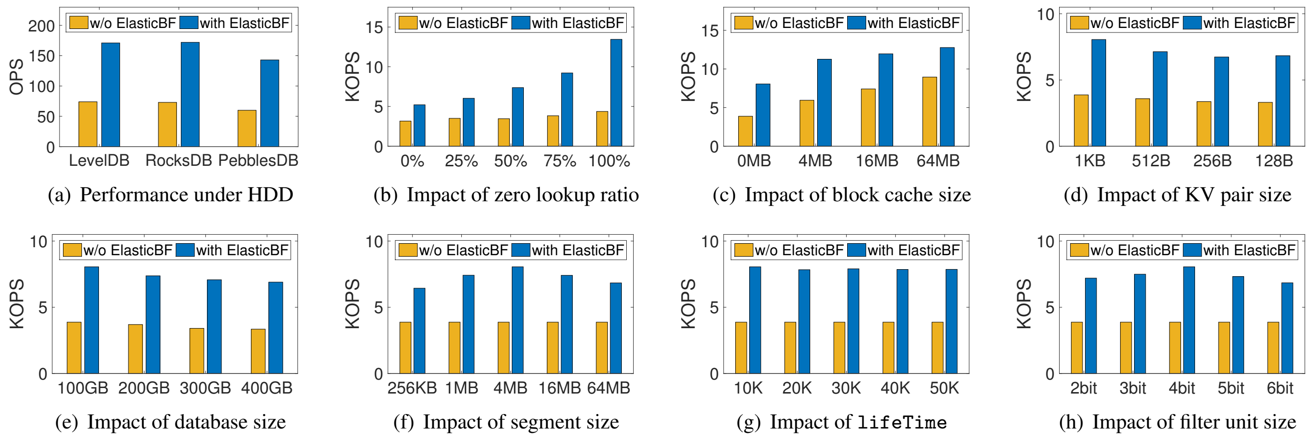
We first evaluate the performance of micro-benchmarks. We conduct the evaluation by issuing 10M Get requests to 100GB KV stores. Figure 14(a) shows the results. We can see that ElasticBF increases the throughput to 1.39× - 2.20× across different workloads. In particular, the improvement ratio increases if the workload is more skewed, and this validates the efficiency of taking into account data locality. However, the performance of Monkey is flat across workloads, the reason is that each zero lookup will traverse all levels of the LSM-tree regardless of the access skewness, thus the number of I/Os caused by each Get request due to false positive keeps almost the same under the static setting in Monkey. To further justify, we also count the total number of I/Os issued by Get operations. As shown in Figure 14(b), ElasticBF reduces the number of I/Os by 36.9% - 80.9% compared with Monkey. We then study the impact of zero lookup ratio, and Figure 14(c) shows the result by fixing the Zipfian constant as 0.99. ElasticBF improves the throughput to 1.16× - 1.39×. In particular, as the number of I/Os to find the relevant items accounts for a higher proportion when there are fewer zero lookups, ElasticBF results in smaller benefits in the case of lower zero lookup ratio. Finally, we also study the performance impact on 400GB KV stores, as shown in Figure 14(d), ElasticBF preserves similar improvement, e.g., it increases the throughput to 1.36× - 2.15×.



(a) Throughput with Zipf 0.99 (b) Throughput with Zipf 1.2

Figure 15: Compared with Monkey under YCSB benchmarks.

Next, we study the performance of ElasticBF under the YCSB benchmarks with four threads, each workload of YCSB issues 10M operations on a 100GB KV store. To consider the impact of access skewness, we conduct two sets of experiments by setting the Zipfian constant as 0.99 and 1.2, respectively. Figure 15(a) and Figure 15(b) show the results. We can see that ElasticBF outperforms Monkey for all read-dominated workloads. In particular, for Workload C (100%



(a) Performance under HDD
(b) Impact of zero lookup ratio
(c) Impact of block cache size
(d) Impact of KV pair size
(e) Impact of database size
(f) Impact of segment size
(g) Impact of `lifeTime`
(h) Impact of filter unit size

Figure 16: Read throughput under different system and parameter configurations.

reads) and Workload D (95% reads and 5% inserts), when the Zipfian constant is 0.99, ElasticBF obtains 1.29× and 1.20× throughput, respectively, and the improvement ratios increase to 1.99× and 1.89× when the workload becomes more skewed. For other workloads, such as Workload A, B and F, the improvement of ElasticBF is small, this is because the updates in these workloads make most of the keys requested by Get be at lower levels, and Monkey can gain the benefits even with a level-based static scheme, and thus the further improvement of ElasticBF is small.

ElasticBF performs better than Monkey mainly because of the finer-grained hotness identification strategy, i.e., the segment-level hotness identification and Bloom filter adjustment. We will further evaluate the impact of the granularity of hotness identification in details by varying the segment size in the next subsection. Besides the performance improvement, ElasticBF is also a more general Bloom filter management scheme, e.g., it can be deployed in KV stores which use complicated compaction strategies [15, 18], while in these cases, the LSM-tree structure becomes hard to predict, so it is hard for Monkey to find the optimal setting of Bloom filters, and Monkey also fails to dynamically adjust the setting when the structure of the LSM-tree changes.

4.5 Impact of Different Configurations

In this subsection, we study the performance impact of different configurations on ElasticBF. We use one thread to run YCSB benchmark, and issue 10M Get requests to a 100GB database. Due to page limit, we only show the performance improvement over LevelDB, and improvements over RocksDB and PebblesDB are also similar.

Performance under HDD. Figure 16(a) shows the read throughput under HDD. We can see that ElasticBF achieves 2.31× - 2.38× better throughput for LevelDB, RocksDB and PebblesDB. Besides, we also note that the improvement ratio is similar to that under SSD. That is, ElasticBF can improve the read performance of KV stores for all the main-stream storage devices, including both SSD and HDD.

Impact of zero lookup ratio. Figure 16(b) studies the impact of different ratios of non-existent items. Specifically, the x-axis represents the ratio of Get requests which access non-existent keys, i.e., zero lookup. Specifically, ElasticBF increases the read throughput to 1.65× - 3.08× when the zero lookup ratio increases from 0% to 100%. Besides, we can see that with the increasing of zero lookup ratio, ElasticBF achieves larger improvement. This is because looking up non-existent keys leads to the search of all levels in the KV store, and so it needs to check more SSTables. Thus, optimizing the Bloom filter can bring larger benefit to reduce the I/Os, so ElasticBF can get a larger improvement from zero lookups.

Impact of block cache size. We study the impact of cache size. As mentioned before, we enable direct I/O in the experiments. Figure 16(c) shows the results with different block cache sizes, note that the default block cache size is 8MB [17], and we set its size as 64MB which is slightly larger than the amount of Bloom filters (50MB). We can see that with a larger block cache, the read performance improves. Besides, ElasticBF still improves the performance when cache is enabled, e.g., it increases the throughput to 1.43× - 2.08×.

Impact of KV pair size and database size. Figure 16(d) studies the impact of KV pair size. ElasticBF increases the read throughput for different KV pair sizes, e.g., it increases the throughput to 1.99× - 2.08× when we vary the KV pair size from 1KB to 128B. Similarly, Figure 16(e) shows that ElasticBF consistently increases the read throughput for large databases, e.g., it increases the throughput to 2.00× - 2.08× when we vary the database size from 100GB to 400GB.

Impact of segment size, `lifeTime` length and filter unit size. Finally, we study the impact of configuration parameters on the read performance of ElasticBF. First, we consider the impact of segment size, and Figure 16(f) shows the read throughput versus the segment size. Note that the SSTable size is 64MB, if the segment size is also set as 64MB, then it means that we measure hotness and adjust Bloom filters in unit of an SSTable. The results show that the improvement is the largest under the 4MB setting, e.g., the throughput under

the 4MB setting is 17.8% higher than that under the 64MB setting. The reason is that as the segment size decreases, ElasticBF can perform a finer-grained hotness recognition, so it can gain more benefits from the adjustment of Bloom filters. This also demonstrates the effectiveness of the finer-grained design in ElasticBF. However, if the size of each filter unit is too small (e.g., less than a block size), then each load of the filter unit is wasting for the I/Os, so the throughput drops. Second, from the results in Figure 16(g), which show the impact of `lifeTime` length, we can see that and the performance improvement has no big difference. That is, ElasticBF is not sensitive to the `lifeTime` parameter, e.g., we can simply set `lifeTime` according to the total number of segments. Finally, Figure 16(h) shows the impact of filter unit size, the x-axis represents the `bits-per-key` of each filter unit, and we configure the total `bits-per-key` of a filter group as 24 (or 25 for 5bit). The improvement is the largest when using 4bit. This is because if each filter unit uses fewer `bits-per-key`, then it can have more filter units for each SSTable, and this implies to have more hotness categories, but too many categories will also require more I/Os to load enough filter units to achieve low false positive rate.

To summarize this section, we find that ElasticBF can effectively boost the read performance of various KV stores under different storage mediums and database scales, but it mainly focuses on the memory-constrained environment. That is, if the memory capacity is not a bottleneck, then one can simply allocate more bits to each Bloom filter and keeps all of them in memory, in this case, the false positive rate can be very small and the benefit of ElasticBF is limited.

5 Related Work

In recent years, many studies have proposed new designs based on LSM-tree [30]. WiscKey [27] reduces the compaction I/Os by using key-value separation technique to manage keys and metadata in LSM-tree, while stores values into an appended-only log. HashKV [7] further optimizes the value management for key-value separation based design by using hash-based data organization. LSM-trie [42] focuses on small key value pairs, and organizes data into a hash-based trie structure to reduce write amplification. bLSM [35] uses a new merge scheduler to reduce the impact of the compaction on the front-end write performance, and also uses Bloom filters to help efficient lookup. TRIAD [5] reduces write I/Os by leveraging the skewed data popularity and delayed compaction strategy. PebblesDB [32] reorganizes the storage layout inspired from skip lists, thereby avoiding data rewriting in the same level to reduce the compaction overhead. We point out that these works mainly focus on improving the write performance, and they still follow the basic structure of LSM-tree and require Bloom filter, so our work is orthogonal to them, and can be used to further improve the read performance by adaptively adjusting Bloom filters.

Some other studies aim to better utilize the features of emerging storage devices to improve the performance of KV stores. For example, RocksDB [16] utilizes the parallelism of SSDs by scheduling multiple compaction operations concurrently. LOCS [40] leverages the multi-channel of SSDs to exploit the abundant parallelism for efficient compaction and data access. NVMKV [28] cooperates with FTL by mapping KV pairs in physical address space to decrease the redundant work between the store layer and the device layer. HiKV [43] also leverages NVRAM by using a hybrid index. In contract, ElasticBF mainly focuses on the Bloom filter design, and it can improve the read performance on both HDDs and SSDs.

At last, there are also several works considering Bloom filter optimization. In particular, RocksDB [15] uses prefix Bloom filter to reduce read amplification on range queries. SuRF [44] is based on a succinct data structure to reduce I/Os by filtering requests of point queries and range queries. Heterogeneous Bloom filter design is also considered to configure different Bloom filters for different levels or files [10, 45]. However, Monkey [10] adopts a coarse-grained scheme which allocates the same number of filters for SSTables within the same level and also fails to dynamically adjust according to hotness. ElasticBF further leverages the access locality in a finer granularity with dynamical adjustment, and our extensive experiments demonstrate its benefit, especially for skewed workloads. Finally, compared with our previously published workshop paper [45], we also make multiple novel optimizations: (1) we develop a fine-grained heterogeneous scheme by further differentiating segments within each SSTable, (2) we propose a hotness inheritance scheme to quickly obtain the accurate hotness information of the newly generated SSTables during compaction, (3) we implement ElasticBF on top of various KV stores and conduct extensive experiments to demonstrate its efficiency and generality, and (4) we also leverage multi-threading and parallel I/Os in the implementation for performance optimization.

6 Conclusion

In this paper, we developed a fine-grained heterogeneous Bloom filter management scheme called ElasticBF by leveraging the access skewness within workloads. ElasticBF measures the hotness information with a lightweight method and also supports dynamic adjustment of Bloom filters at a fine granularity. As a result, ElasticBF can greatly reduce the expected overall false positive rate without increasing the volume and memory overhead of Bloom filters, and thus speeds up the read performance in KV stores. Finally, we also conducted extensive experiments to demonstrate the efficiency of ElasticBF by building it atop of various KV stores.

Acknowledgements

The work was supported by National Key R&D Program of China under Grant No. 2018YFB1003204, National Nature Science Foundation of China (61772484, 61832011, and 61772486).

References

[1] Apache. Cassandra. http://cassandra.apache.org/.

[2] Apache. Tuning Bloom filters. http://cassandra.apache.org/doc/4.0/operating/bloom_filters.html, 2018.

[3] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013.

[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 Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[5] Oana Maria Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX ATC 17*, 2017.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[7] Helen HW Chan, Yongkun Li, Patrick PC Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] 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. ACM, 2010.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[13] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[14] 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 *ACM SIGOPS operating systems review*. ACM, 2007.

[15] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, and Tony Savor. Optimizing Space Amplification in RocksDB.

[16] Facebook. RocksDB. http://rocksdb.org/.

[17] Facebook. Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache, 2017.

[18] Facebook. Universal Compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction, 2017.

[19] Facebook. Direct IO. https://github.com/facebook/rocksdb/wiki/Direct-IO, 2018.

[20] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. InnoDB Database Forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010.

[21] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[22] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook Messages Case Study. In *FAST*, 2014.

[23] Yangwook Kang, Rekha Pitchumani, Thomas Marlette, and Ethan L Miller. Muninn: A Versioning Flash Key-Value Store Using an Object-based Storage Model. In *Proceedings of International Conference on Systems and Storage*, pages 1–11. ACM, 2014.

[24] Adam Kirsch and Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *ESA*, volume 6, pages 456–467. Springer, 2006.

[25] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's Key-Value Storage System for Cloud Data. In *Mass Storage Systems and Technologies*

(MSST), 2015 31st Symposium on, pages 1–14. IEEE, 2015.

[26] Guanlin Lu, Young Jin Nam, and David HC Du. BloomStore: Bloom-Filter based Memory-efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[27] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *FAST*, pages 133–148, 2016.

[28] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *HotStorage*, 2014.

[29] MongoDB. MongoDB. `https://www.mongodb.com/`.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[31] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *USENIX ATC*, pages 537–550, 2016.

[32] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[33] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[34] J. REN. YCSB-C. `https://github.com/basicthinker/YCSB-C`, 2015.

[35] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012.

[36] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *FAST*, pages 17–30, 2013.

[37] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[38] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[39] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[40] 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*, page 16. ACM, 2014.

[41] WiredTiger. WiredTiger. `http://www.wiredtiger.com/`.

[42] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *USENIX ATC 15*, pages 71–82. USENIX Association, 2015.

[43] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[44] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336. ACM, 2018.

[45] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *USENIX HotStorage 18*, 2018.

[46] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores

Oana Balmau
University of Sydney

Florin Dinu
University of Sydney

Willy Zwaenepoel
University of Sydney

Karan Gupta
Nutanix Inc.

Ravishankar Chandhiramoorthi
Nutanix Inc.

Diego Didona
IBM Research – Zurich

Abstract

LSM-based KV stores are designed to offer good write performance, by capturing client writes in memory, and only later flushing them to storage. Writes are later compacted into a tree-like data structure on disk to improve read performance and to reduce storage space use. It has been widely documented that compactions severely hamper throughput. Various optimizations have successfully dealt with this problem. These techniques include, among others, rate-limiting flushes and compactions, selecting among compactions for maximum effect, and limiting compactions to the highest level by so-called fragmented LSMs.

In this paper we focus on latencies rather than throughput. We first document the fact that LSM KVs exhibit high tail latencies. The techniques that have been proposed for optimizing throughput do not address this issue, and in fact in some cases exacerbate it. The root cause of these high tail latencies is interference between client writes, flushes and compactions. We then introduce the notion of an I/O scheduler for an LSM-based KV store to reduce this interference. We explore three techniques as part of this I/O scheduler: 1) opportunistically allocating more bandwidth to internal operations during periods of low load, 2) prioritizing flushes and compactions at the lower levels of the tree, and 3) preempting compactions.

SILK is a new open-source KV store that incorporates this notion of an I/O scheduler. SILK is derived from RocksDB, but the concepts can be applied to other LSM-based KV stores. We use both a production workload at Nutanix and synthetic benchmarks to demonstrate that SILK achieves up to two orders of magnitude lower 99th percentile latencies than RocksDB and TRIAD, without any significant negative effects on other performance metrics.

1 Introduction

Latency-critical applications require data platforms that are able to deliver low latency and predictable throughput. Tail latency is especially important, because applications often exhibit high fan-out queries whose overall latency is determined by the response time of the slowest reply. Log-structured merge key-value stores (LSM KVs), such as RocksDB [18], LevelDB [14] and Cassandra [30], are widely adopted in production environments to provide storage beyond main memory for such latency-critical applications, especially for write-heavy workloads. At Nutanix, we use LSM KVs for storing the meta-data of our core enterprise platform, which serves thousands of customers with petabytes of storage capacity.

KV stores support a range of client operations, such as `Get()`, `Update()` and `Scan()`, to store and retrieve data. LSM KVs strive for good update performance by absorbing updates in an in-memory buffer [36, 37]. A tree-like structure is maintained on storage. In addition to client operations, LSM KVs implement two types of *internal operations*: *flushing*, which persists the content of in-memory buffers to disk, and *compaction*, which merges data from the lower into the higher levels of the tree.

In this paper we demonstrate that tail latencies in state-of-the-art LSM KVs can be quite poor, especially under heavy and variable client write loads. We introduce the notion of an *I/O scheduler for LSM KVs*. We implement this I/O scheduler in RocksDB, and we show up to two orders of magnitude improvements in tail latency.

Our work complements much recent work that has sought to improve the client throughput of LSM KVs (e.g., [4, 24, 28, 32, 34, 38, 39, 42, 43]). Client throughput is improved by reducing the cost of internal operations, but this does not suffice to reduce tail latency. Internal operations remain necessary, and client operations that arrive during ongoing internal operations experience increased latency because of interference with these internal tasks. The internal operations may be fewer in number and less costly, reducing the probability of latency spikes, but in practice they occur sufficiently often to influence the higher order percentiles of the latency distribution, especially if client load is bursty.

One may at first think that limiting the I/O bandwidth allocated to internal operations, as is commonly done in production systems, would avoid latency spikes due to interference between internal work and client load. On closer inspection, however, we find this not to be the case. As a simple example, consider a burst of client writes, triggering a burst of flushes. If a number of compactions is going on at the same time, the flushes have to share the limited bandwidth with the compactions, and they become slow. This leads to the in-memory component filling up and blocking further writes, hence producing latency spikes. Limiting the rate of compactions is also insufficient, because they can lead to the lowest level of the tree filling up, stalling flushes, and in turn stalling writes.

These and other observations lead us to the conclusion that reducing the cost of internal operations or limiting their bandwidth allocation does not suffice to avoid latency spikes, and that instead there is a fundamental need for coordination between the client load and the load imposed by different internal operations. To this end we introduce an *I/O scheduler* for an LSM-based KV store.

We build a new KV store, SILK, which we derive from RocksDB. The I/O scheduler in SILK (1) dynamically allocates bandwidth between client and internal operations, (2) gives preference to internal operations that may block client operations, and (3) allows preemption of less critical internal operations. Other techniques could possibly be included, but we have found these sufficient to get two orders of magnitude benefits in tail latency for write-heavy workloads, with no negative effects on throughput or average latency. Also, SILK does not produce significant negative effects in read- and scan-heavy workloads.

Contributions. The main contributions of this paper are:

1. An extensive empirical study that demonstrates the high tail latencies of current LSM KVs.
2. The introduction of an I/O scheduler for LSM KVs, and various scheduling techniques useful for reducing tail latency while maintaining good throughput.
3. An implementation of an LSM KV store I/O scheduler in an industry-standard LSM KV store (RocksDB).
4. An experimental evaluation demonstrating up to two orders of magnitude improvements in tail latency in our production workload, without significant negative effects on other performance metrics or workloads.

2 LSM KV background

2.1 LSM KV architecture

An LSM KV store has three main components: the memory component, the disk component, and the commit log.

Memory component. The memory component C_m is a sorted data structure that resides in main memory. Its size is typically small, around a few tens of MBs. The purpose of C_m is to temporarily absorb user updates. When C_m fills up, it is replaced by a new, empty component. The old memory component is then in the background flushed as is to level 0 (L_0) of the LSM disk component.

Disk component. The disk component C_{disk} is structured into multiple levels (L_0, L_1, . . . L_n), where each level is larger than the previous (lower) level by a configurable factor (e.g., 10). Each level contains multiple sorted files, called SSTables. The number of SSTables on a given level is limited by a configuration parameter, as is the maximum size of an individual file for a given level. SSTables on levels L_i $(i > 0)$ have disjoint key-ranges. L_0 allows overlapping key-ranges between files.

Commit log. The commit log C_{log} stores the updates that are made to C_m (in small batches) on stable storage. C_{log} is usually a few hundreds of MBs large. It is used if the application requires the data to be recoverable in case of a failure, but it is not mandatory. The techniques we propose in SILK apply regardless of whether C_{log} is active or not.

2.2 LSM KV operations

LSM KVs implement two main kinds of operations, which are executed by disjoint thread pools.

Client operations. The main client operations in LSM KVs are writes *(Update(k, v))*, reads *(Get(k))*, and range scans *(Scan(k1, k2))*. *Update(k, v)* associates value v to key k. Updates are absorbed in C_m, to achieve high write throughput. *Get(k)* returns the most recent value of k. The read first goes to C_m. If k is not found in C_m, the read continues to L_0, L_1, . . . L_n, until k is found. By design, at most one SSTable is checked on each level L_i for $i > 0$. On the contrary, more than one SSTable in L_0 may need to be checked because L_0 SSTables may contain the entire key-range. Per-SSTable Bloom filters [7, 25] are used to address this issue. Therefore, in practice, only one SSTable ends up being checked on L_0 most of the time. *Scan(k1, k2)* returns a range of key-value tuples with the keys ranging from $k1$ to $k2$. First, C_m is queried for keys in the $k1$–$k2$ range. Then, SSTables in C_{disk} that may contain the $k1$–$k2$ range are read, going down the levels, until all the keys are found. Client operations are enqueued and served in FIFO order by a fixed-size worker thread pool.

Internal operations. LSM KVs implement *flushing* and *compaction* as background processes. Flushing writes C_m as is to L_0. Because flushing speed affects the rate at which new memory components can be installed, memory components are written to disk without additional processing. As a result, L_0 allows overlapping key-ranges between files. Compaction is the operation that cleans up the LSM tree. It merges SSTa-

bles in level L_i of C_{disk} into SSTables with overlapping key-ranges in L_{i+1}, discarding older values in the process. When the size of L_i exceeds its maximum, an SSTable F in L_i is picked and merged into the SSTables in L_{i+1} that have overlapping key-ranges with F, in a way similar to a merge sort. Most LSM KVs support parallel compactions, apart from compactions from L_0 to L_1 which are not parallelized because of overlapping key-ranges on L_0. Compaction induces large I/O overhead by reading the SSTables and writing the new ones to disk.

The system maintains an internal FIFO work queue, where flushes and compactions are enqueued. When a new internal work request is enqueued, it is placed at the end of the internal work queue. A flush is enqueued when C_m fills up. A compaction operation may be enqueued after a flush completes, or after a compaction completes. A pool of internal worker threads serve the requests in the internal work queue. In current LSM KVs an internal operation is enqueued whenever the system deems it necessary in order to maintain the structure of the LSM tree (e.g., when the maximum size or maximum number of files is reached on a level).

2.3 State-of-the-art LSM-based systems

Our experimental study includes three state-of-the-art systems: RocksDB, TRIAD, and PebblesDB.

RocksDB [18] developed at Facebook, is a popular system in production environments, including ours. Its architecture follows the description above. In addition, RocksDB provides a rate limiter to restrict the I/O bandwidth for internal operations. The bandwidth can be set to a fixed value, or RocksDB can change it over time in a multiplicative-increase, multiplicative-decrease manner [19]. This *auto-tuned* version of the rate limiter adapts the bandwidth to the amount of internal work, allocating more bandwidth when there is more pending work.

TRIAD [4] reduces the overhead of internal operations through three techniques. First, TRIAD keeps frequently updated keys in C_m, decreasing internal operation overhead in skewed workloads. Second, TRIAD provides an improvement to the flushing operation, by leveraging data already written in C_{log}. Finally, at the disk level, TRIAD employs a cost-based approach to trigger compaction from L_0 to L_1. Compaction happens only when there is significant key-range overlap, reducing the frequency of compaction operations and amortizing their cost.

PebblesDB [39] avoids most of the compaction overhead of merging and rewriting SSTables, by allowing overlapping key-ranges on all but the highest tree level through the use of Fragmented LSM trees. PebblesDB orders SSTables by key-ranges on each level, and uses special pointers called *guards* to indicate where a given key-range is on a level. When the number of guards on a level reaches a threshold, the guards and the corresponding keys are moved to the next level, mostly without re-writing the SSTables. PebblesDB only requires compaction at the highest tree level, when the number of guards reaches a threshold.

3 Performance requirements for LSM KVs

LSM KVs should meet the following requirements:

1. **Low tail latency.** In environments where LSM KVs serve applications with high fan-out operations, in which the slowest reply within an operation determines the latency of the whole operation, low tail latency is a key requirement [15].

2. **Predictable throughput.** LSM KVs must deliver a throughput that matches the client load *at any time*. Throughput variability is a well-known problem in LSM KVs, mostly stemming from the interference between LSM internal work and client requests.

3. **Small main memory footprint.** Typically, LSM KVs are only one piece of a wider set of services that are accessed by an application. For example, a KV store that handles meta-data can co-exist on the same machine with other services that require large amounts of memory, making memory a constrained resource.

LSM issues. A common issue in LSM KVs is interference between LSM internal work and client operations when a sudden burst of client-side writes occurs in parallel with long-running, resource-intensive compaction tasks. Despite the fact that internal LSM work directly influences client latency, it is handled without being aware of the client load. For instance, large compactions (e.g., compacting tens of GBs) may occupy a large fraction of the I/O bandwidth for extended periods of time (e.g., tens of minutes). The resulting problem is two-fold. First, when flushes do not proceed in a timely manner, the memory component fills up, and incoming writes cannot be absorbed in the memory component. Second, slow L_0 to L_1 compactions lead to the accumulation of a large number of SSTables on L_0. In extreme situations, when the maximum number of SSTables on L_0 is reached, this dynamic brings the entire system to a halt. Both scenarios lead to severe spikes in client latency.

4 Experimental study of tail latency

We perform an extensive experimental study to show that established techniques used in industry and state-of-the-art research systems do not solve the issue of tail latency.

4.1 Experimental environment

We use the YCSB [11] update-intensive workload, corresponding to a 50:50 read:write ratio on 1 KB items (YCSB

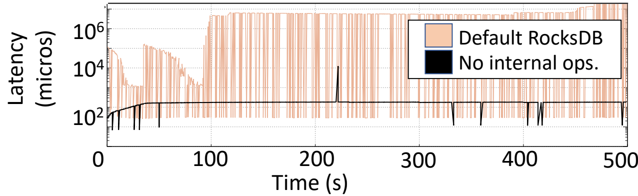


Figure 1: RocksDB compared to a RocksDB version with no internal operations. Internal operations lead to spikes in client request 99th percentile latency.

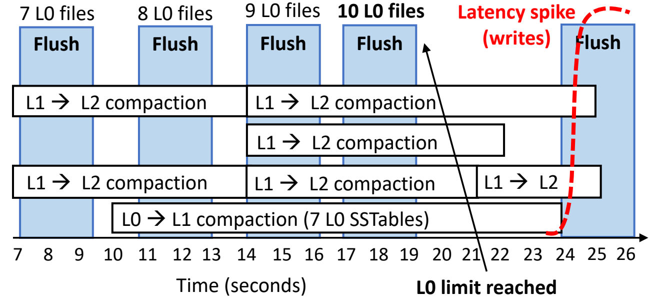


Figure 2: RocksDB. Timeline of internal operations during a writes latency spike (dashed red line) caused by L_0 reaching full capacity. L_0 reaches 10 SSTables at $t = 19$, so flushes are temporarily paused.

workload A), with a uniform data distribution. We choose this workload because it is representative for write-intensive workloads in LSM KVs. Furthermore, a uniform workload allows us to detect more quickly performance problems that in skewed workloads would remain hidden due to in-memory caching [4]. LSM KVs are notorious for having numerous tuning knobs. Throughout this study, we configure all the involved systems to the best of our abilities, following guidelines in [23, 39]. The hardware configuration we use in this study is described in Section 6.1.

Each experiment consists of a population phase followed by read and write operations issued at 18 Kops/sec. Unless stated otherwise, all experiments are run without I/O bandwidth rate limiting for internal operations. For RocksDB, unless stated otherwise, we use two memory components of 128MB each. For the rest of the systems we limit the memory use to 1GB. We configure the maximum number of concurrent internal operations to ten for all systems. The latency is measured every second. The 99th percentile latency is computed for every 1-second interval.

4.2 RocksDB

We first show the performance degradation of client operations caused by internal operations in RocksDB. To this end, we compare RocksDB with a modified version of RocksDB in which compaction and flushing are disabled. We disable internal operations by discarding C_m when it fills up (the data store is pre-populated with the full set of keys, so persistent storage is accessed by reads, if necessary).

Figure 1 shows the performance of the two systems over time. The 99th percentile latency of operations in RocksDB is 2 to 4 orders of magnitude higher than in the system without internal operations. These spikes are not present at the 50th and 90th percentiles of the latency distribution. Both reads and writes experience latency spikes at the same time and of the same magnitude, despite their different access paths in the LSM KV store (i.e., writes complete in-memory, while reads are typically served from persistent storage).

The main culprit for the latency spikes is the fact that writes get blocked by virtue of C_m filling up. The reads then get queued behind these writes in the threading architecture.

We identify two main reasons for write latency spikes, illustrated in Figures 2 and 3. The examples showcase real scenarios encountered while profiling RocksDB.

Figure 2 illustrates an example of a write latency spike (the red dashed line) occurring because L_0 reaches maximum capacity (10 SSTables in this example). Several compactions on levels L_0, L_1 and L_2 occur in parallel between $t = 14$ and $t = 23$. Even if many parallel compactions can run at higher levels (i.e., L_i to L_{i+1}, where $i > 0$), there can only be one L_0 to L_1 compaction running at a time. Since I/O bandwidth is spread equally over all compactions, L_0 to L_1 compaction is slowed down. Consequently, L_0 is not cleared fast enough, which, in turn, causes flushes from C_m to be temporarily halted.

A second cause for latency spikes is C_m filling up because of slow flushing, as illustrated in Figure 3. Here, L_0 does not fill up, reaching only 7 SSTables at $t = 5$. However, the flush starting at $t = 0$ takes an unusually long time (5 seconds compared to 1-2 seconds for a typical flush). The cause is that, by coincidence, a large number of compactions are running at the same time, which makes flushing slow because of limited available I/O bandwidth. There are 7 ongoing compactions at the time of the very slow flush.

4.3 Rate-limited RocksDB

Limiting the I/O bandwidth for internal operations is a popular technique to prevent them from consuming an excessive amount of I/O bandwidth, and hence to "shelter" client operations. We now report the results that we obtain when running RocksDB with limited I/O bandwidth for internal operations [22].

Figure 4 shows the 99th percentile latency of client operations over time when limiting the I/O bandwidth for internal operations to 50, 75 and 90 MB/s. For the sake of legibility, in Figure 4 we show the results of the experiment only up to the time that the tail latency greatly deteriorates (at 900s for 50MB/s, etc). The higher the bandwidth assigned to inter-

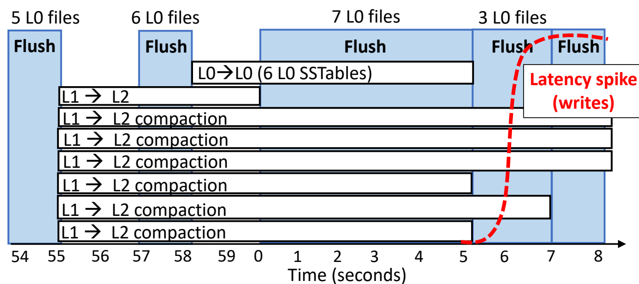


Figure 3: RocksDB. Timeline of internal operations during a writes latency spike caused by slow flushing. From $t = 0$ to $t = 7$, flushes are slowed down by many parallel L_1 to L_2 compactions monopolizing I/O bandwidth. Consequently, C_m fills up, not being able to absorb incoming updates.

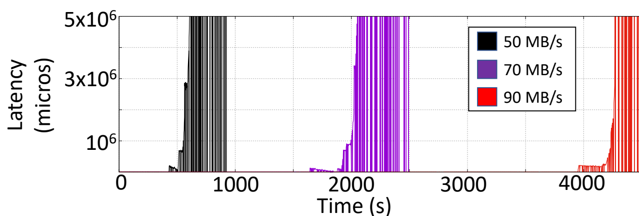


Figure 4: RocksDB. 99th percentile of client request latency when limiting the I/O bandwidth for internal operations.

nal operations, the longer the system is able to postpone the occurrence of latency spikes. However, restricting the bandwidth for internal operations results in slowing them down. This approach therefore increases the likelihood that at some later point many compactions are running at the same time, contending for the limited I/O bandwidth.

4.4 RocksDB with increased C_m

We investigate whether allocating larger memory buffers influences tail latencies in LSM KV. To this end, we run a series of experiments in RocksDB where we increase the total size of the memory component(s) to 1GB – a value close to the upper limit of what we can allow in our production environments (see Section 3). We distribute the memory first into two 500MB memory components and then into ten memory components of 100MB each. We also vary the maximum number of flushes, experimenting with one and ten parallel flushing threads. We find the setup with ten memory components and a single flushing thread to be the most efficient in postponing the latency spikes because the memory absorbs more updates and the data flow from memory to L_0 matches more closely the L_0 to L_1 compaction flow. However, we encounter tail latency spikes sooner or later in all of these cases, for similar reasons to the ones in the scenarios above.

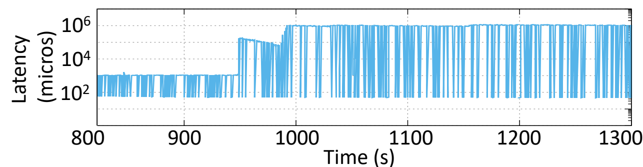


Figure 5: TRIAD. 99th percentile latency. Despite reducing internal operations overhead, TRIAD does not prevent latency spikes during resource-intensive compactions.

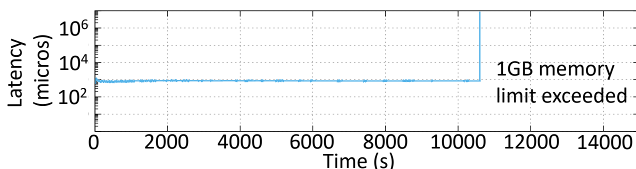


Figure 6: PebblesDB. 99th percentile latency. Postponing compaction keeps the latency low. Experiment ends because of high memory overhead.

4.5 TRIAD

Reducing the overhead of internal operations, as done by state-of-the-art systems [4, 12, 13, 17], is not enough to avoid resource interference. We use TRIAD [4] as a representative of such state-of-the-art systems in the next experiment. Figure 5 shows the 99th percentile latency of client operations over time. In this scenario, TRIAD reduces compaction overhead mainly by choosing when to run L_0 to L_1 compactions depending on key-range overlap. Postponing compactions at the lower levels (closer to C_m) results in postponing compactions at the higher levels. So, in the long term, TRIAD increases the likelihood of running many concurrent compactions. Consequently, the 99th percentile of client operations shows no spikes for the first $\approx$ 1,000 seconds but, after that point, shows frequent and significant spikes.

4.6 PebblesDB

Figure 6 shows PebblesDB's 99th percentile latency over time. The experiment stops after 10,500 seconds. Although we provide PebblesDB with more memory than RocksDB and TRIAD, it runs out of memory at this time. The memory consumption is due to the frequent creation of guards and Bloom filters in a write-intensive workload. During its uptime, PebblesDB provides very good tail latencies due to the absence of compactions. In other words, the LSM tree is restructured through the use of guards but no data compactions occurred.

To create a situation in which PebblesDB experiences compactions, we run it with a read-intensive workload (95:5) which reduces memory pressure. With this workload, tail latencies remain very good in the early going, but after around

8 hours, when compaction sets in, the system effectively comes to a halt. PebblesDB stalls client operations when it has to perform the very resource-demanding compaction on the highest level of the tree. When such compaction takes place, all threads for internal operations are busy, so they cannot push down guards and keys from the lower levels of the tree. Hence, to maintain the tree integrity, PebblesDB stalls client operations until compaction terminates.

4.7 Lessons learned

We gain three main insights from our experimental study.

Lesson 1) The main reason for high tail latency is the fact that writes get blocked by C_m filling up. There are two principal reasons for this. The first reason is that L_0 on disk is full, which causes flushes from C_m to be halted. L_0 reaches its capacity if L_0 to L_1 compaction cannot keep up. The second reason is that, by coincidence, a large number of compactions are happening concurrently, which causes flushing to be slow because of limited available bandwidth.

Lesson 2) Simply limiting bandwidth for internal operations does not solve the problem of limited bandwidth being available for flushes and can in fact exacerbate it in the long run. This approach effectively postpones compactions, and therefore increases the likelihood that at some later point many compactions occur at the same time.

Lesson 3) Recent approaches to improve throughput, such as being selective about starting compactions or only performing compactions at the highest level, avoid latency spikes in the short run, but aggravate the problem in the long run, because they too increase the likelihood of many concurrent compactions at some later point in time.

As a corollary to Lesson 1, we conclude that not all internal operations are equal. Internal operations on the lower levels of the tree (i.e., closer to C_m) are critical, because failing to complete them in a timely fashion may result in stalling client operations.

Finally, as a corollary to Lessons 2 and 3, it is essential to run performance tests for an extended amount of time, lest these issues go undetected.

5 SILK

5.1 SILK design principles

SILK integrates the lessons we learn from our experimental study into an *I/O scheduler* for internal and external work. SILK follows three core design principles.

1) Opportunistically allocating I/O bandwidth to internal operations. SILK leverages the fact that, in production workloads, the load of client-facing operations typically varies over time (see Figure 7). SILK allocates less I/O bandwidth to compactions on higher levels during peak

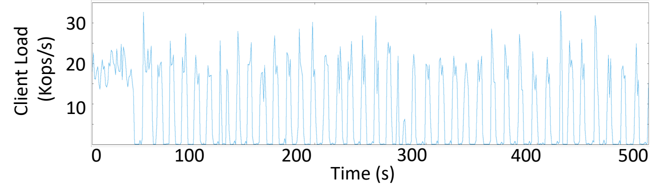


Figure 7: Client load in Nutanix production workload. Real workloads are not flat lines.

client load, and exploits transient low-load periods to boost the processing of internal operations. Dynamic I/O throttling enables SILK (1) to limit interference between internal operations and client-facing ones, and (2) to avoid accumulating over time too large a backlog of internal work, preventing overload conditions in the long term.

2) Prioritizing internal operations at the lower levels of the tree. SILK integrates Lesson 1 in its design by introducing prioritized execution of flushes and compactions from L_0 to L_1. SILK splits internal operations of LSM KVs into three categories with respect to the effect they have on client latencies: (1) SILK ensures that the flushes are fast, making room in memory to absorb incoming updates, which directly affects write latency, (2) SILK gives second priority to L_0 to L_1 compactions, ensuring that L_0 does not reach its full capacity, so that flushes can proceed, (3) SILK gives third priority to compactions on the levels below L_1 because, while they maintain the structure of the LSM tree, their timely execution does not significantly affect client operation latencies in the short term.

3) Preempting compactions. SILK implements a new compaction algorithm that allows internal operations on lower levels of the tree to preempt compactions on higher levels.

5.2 SILK implementation

5.2.1 Opportunistically allocating I/O bandwidth

SILK continuously monitors the bandwidth used by client operations and allocates the available leftover I/O bandwidth to internal operations. The client load monitoring and rate limiting are handled by a separate SILK thread. The monitoring granularity is a system parameter which depends on the frequency of fluctuations in the workload; the monitoring granularity in SILK is currently configured to 10 ms.

If the total I/O bandwidth available to the LSM KV store is T B/s, SILK measures the bandwidth C B/s used by the client requests and it continuously adjusts the internal operation bandwidth to $I = T - C - \varepsilon$ B/s , where ε is a small buffer. To adjust the I/O bandwidth, SILK makes use of a standard rate limiter (e.g., [22]). SILK maintains a minimum configurable I/O bandwidth threshold for flushing and L_0 to L_1 compactions, because these operations directly influence client latency.

To minimize overhead associated with changing the rate limit, SILK only adjusts the limit if the difference between the current value and the new measured value is significant. We empirically set this threshold to be 10 MB/s. We find that lower thresholds cause overly frequent changes in the rate limit. The role of ε is to account for small fluctuations in client load which are not significant enough to adjust internal operation bandwidth using the rate limiter.

5.2.2 Prioritizing and preempting internal operations

Recall that in LSM KVs internal work is handled by a pool of internal worker threads. Once a flush or a compaction is completed, the system checks whether more internal work is needed by assessing the size of the levels and the state of the memory component. If needed, more internal work tasks are scheduled in an internal work queue. SILK maintains two internal worker thread pools: a high-priority one for flushing, and a low-priority one for compactions.

Flushing has the highest priority among the internal operations. Flushes have their dedicated thread pool and always have access to the I/O bandwidth available for internal operations. The minimum flushing bandwidth is chosen to be sufficient to be able to flush the immutable memory component before the active one fills up. The current implementation of SILK allows two memory components (i.e., an immutable one, and an active one) and one flushing thread. If memory constraints allow it, having multiple memory components and flushing threads may help sustain longer client activity peaks.

$\mathbf{L_0}$ to $\mathbf{L_1}$ compaction. SILK needs L_0 to L_1 compactions to progress to ensure that there is enough room to flush on L_0. Unlike flushes, these compactions do not have a dedicated thread pool. If L_0 to L_1 compaction needs to proceed and all the threads in the compaction pool are running higher-level compactions, one of them is *preempted*. This way, L_0 to L_1 compactions do not wait behind higher-level compactions. In the current implementation the preempted compaction task is picked at random.

L_0 to L_1 compaction, like all internal operations is subject to dynamic I/O throttling. However, this type of compaction is never paused, even if SILK may choose to give no bandwidth to compactions. In order to keep L_0 to L_1 compaction running, SILK temporarily moves this job to the high priority thread pool and keeps it running via a high priority thread (i.e., same priority as the flush thread). In this case, the minimum flushing bandwidth mentioned above is shared by the flushing thread and the L_0 to L_1 compaction thread. At most one L_0 to L_1 compaction can run at a time, due to consistency issues caused by overlapping key-ranges. So, only one extra thread is added in the high priority thread pool. Recent versions of RocksDB support L_0 to L_0 compactions as an optimization to quickly reduce the number of SSTables on L_0 [21]. Since this optimization is beneficial for allowing flushes to proceed, SILK treats this case the same as L_0 to L_1 compactions.

Higher-level compactions. Compactions on levels higher than L_1 are scheduled in the low priority compaction thread pool. They make use of the I/O bandwidth available, as indicated by the dynamic rate limiter described in Section 5.2.1. SILK can pause and resume these larger compactions, either individually (because of L_0 to L_1 compaction preemption) or at the level of the thread pool (because of high user load).

It might happen that an L_1 to L_2 compaction is invalidated by the work done by an L_0 to L_1 compaction which preempted it. In this case, SILK discards the partial work done by the higher level compaction. We did not find this wasted work to significantly impact performance.

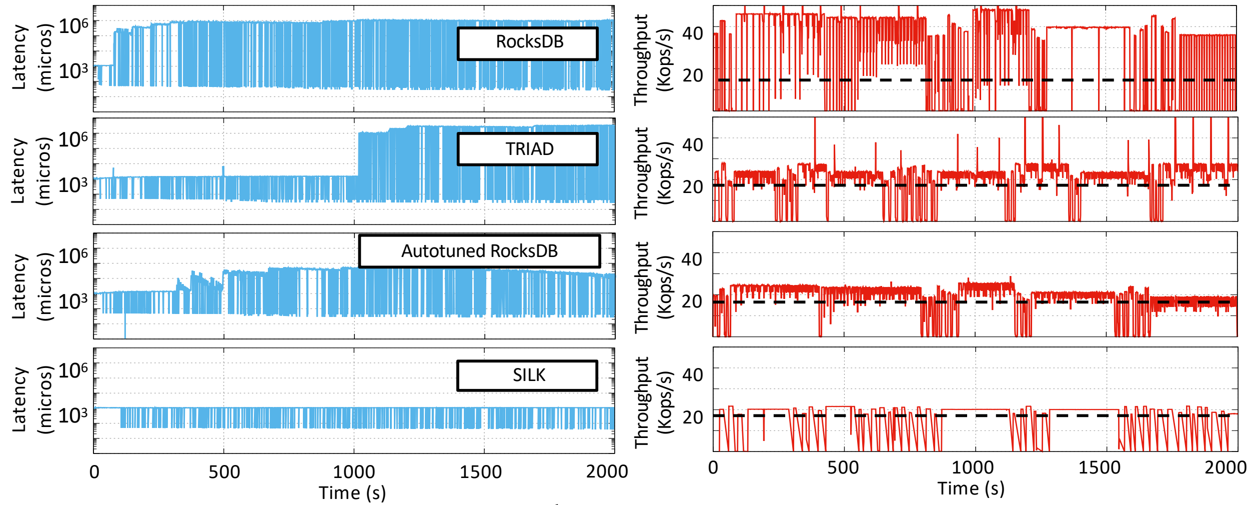
SILK supports parallel compaction, like most current LSM KVs. By default, and assuming equally aggressive compaction threads, each thread gets a similar share of the resources. LSM KVs do not allow parallel compactions from L_0 to L_1, so, if many parallel compactions are allowed, most of the compaction threads are working on the higher levels. This is detrimental to the client operation latencies, since L_0 to L_1 compactions are key to system performance and would benefit from getting the bulk of the resources. Reducing the size of the thread pool, together with SILK's compaction preempting scheme allows each internal worker thread to access a larger share of the resources, which results in faster completion of critical compactions.

The typical recommendation [23] is to set the number of compaction threads equal to the total number of cores. However, we find that the number of compaction threads should instead depend on the total drive I/O bandwidth and the amount of I/O bandwidth required by individual compaction operations. For instance, for a drive with 200MB/s bandwidth, four internal work threads is a suitable choice; even if all the threads happen to run in parallel, they are still allocated a large enough amount of bandwidth to finish the compaction operations fast, thus avoiding scenarios like the ones described in Section 4.

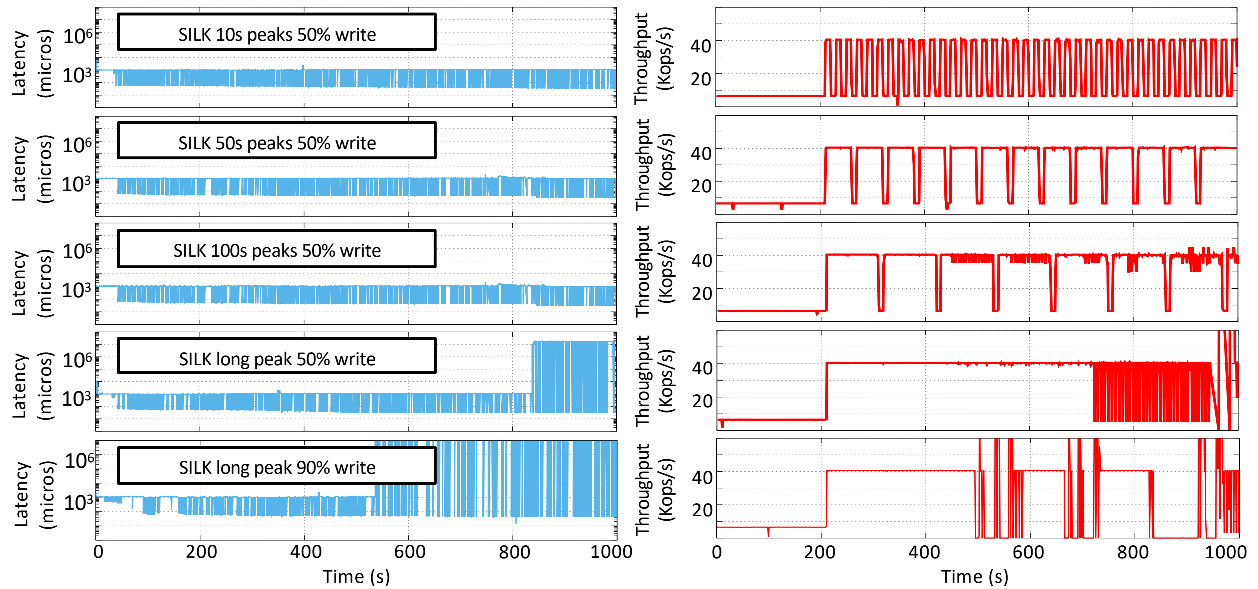
Currently, SILK controls the total I/O bandwidth allocated for compactions in the low priority thread pool. An interesting strategy to explore would be allocating different amounts of bandwidth to compactions at a finer granularity, depending on how urgent the compaction task is considered. We find the current approach to bring good improvements without this additional level of complexity.

6 Evaluation

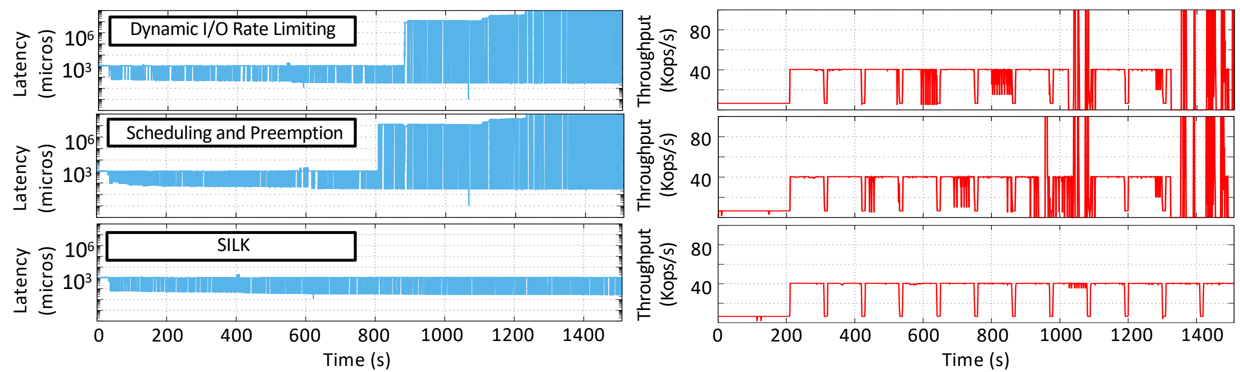
We implement SILK as an extension of RocksDB – used at Nutanix–, and of TRIAD. The source code of SILK is available at `https://github.com/theoanab/SILK-USENIXATC2019`. In what follows, we refer to the



A). Nutanix production workload. Left: 99th percentile latency, log scale on y-axis. Right: throughput. SILK maintains low and steady tail latency and its throughput closely follows the client load. Throughput presents high fluctuations in RocksDB and TRIAD. Average throughput is shown by the dashed black line.



B). Synthetic workloads. Degradation is faster during long peaks, as workloads get more write intensive.



C). Breakdown of SILK techniques. 50% write - 50% read workload, 100s peaks.
SILK's techniques complement each other in order to maintain low tail latency in the long run.

Figure 8: SILK performance in production and synthetic workloads.

RocksDB and the TRIAD extensions as RocksDB-SILK and TRIAD-SILK, respectively. An I/O scheduler could also be applied to PebblesDB, with suitable modifications for the fact that compactions only happen at the highest level. We do not extend PebblesDB with an I/O scheduler because of lack of familiarity with the code base and because PebblesDB's memory demands are not suitable in our environment.

We evaluate SILK with production and synthetic workloads, focusing on write-intensive workloads. We compare against TRIAD and RocksDB and show that:

- SILK achieves up to 2 orders of magnitude lower tail latency than state-of-the-art systems (Section 6.2).
- SILK's performance does not deteriorate over time in long running production workloads (Section 6.2).
- SILK provides stable throughput, close to the client load (Sections 6.2 and 6.4).
- SILK does not create any significant negative side effects on other important metrics such as average latency and read performance (Section 6.3).
- SILK can sustain long client activity peaks interrupted by short client activity lows (Section 6.4).
- The techniques used in SILK contribute to the results above in complementary ways (Section 6.5).

6.1 Experimental setup

Hardware. We perform the evaluation on a 20-core Intel Xeon, with two 10-core 2.8 GHz processors, 256 GB of RAM, and 960GB SSD Samsung 843T. All systems were restricted to run with 1GB of RAM using Linux control groups.

Benchmark. We compare the performance of RocksDB-SILK and TRIAD-SILK to RocksDB, TRIAD, and a version of RocksDB that uses the auto-tuned rate limiter [19]. All experiments are run through `db_bench`, one of RocksDB's standard benchmarking tools [20].

Measurements. Load-generator threads issue requests in an open loop according to the workload characteristics. They deposit the requests in the queues of the KV store worker threads. Latency is measured on the side of the load-generator threads, capturing both *queuing* time and *processing* time. We measure the 99th percentile tail latency and the throughput over one-second intervals (i.e., not cumulative over the entire experiment run). We report throughput and latency every second in the time-series plots.

Dataset. The dataset size for both the production and the synthetic workloads is approximately 500GB. The KV-tuple sizes vary between the production and the synthetic workloads. In all the experiments the data store is pre-populated with the entire dataset.

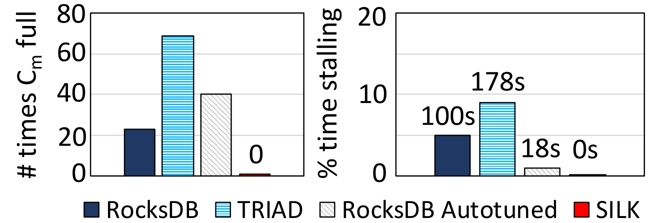


Figure 9: Production workload. Number of times C_m cannot be flushed (left) and time spent stalling writes (right).

KV store configuration. We use a 128MB memory component size and two memory components (i.e., one active and one immutable). In SILK, flushing and L_0 to L_1 compactions proceed at a rate of 50MB/s if SILK paused the other internal operations. The total I/O bandwidth allocated to the LSM KV store is 200MB/s. The `level0-slowdown` and `level0-stop` parameters (used to slow down or stop client writes once a maximum number of files is reached on L_0) are configured to very large values in all data stores so as not to artificially interfere with the measured latency. We use a thread pool of 4 threads for internal operations (including the flushing thread) for all the systems. All systems are pinned to 16 cores, out of which 8 are used by the worker threads, and 8 are used by the internal operations and other LSM threads (e.g., monitoring the client load in SILK). The load-generator threads run on separate dedicated cores.

Compression and commit logging are disabled in all reported measurements. While enabling them affects the absolute performance results, it does not impact the conclusions of our evaluation: the performance differences between RocksDB-SILK or TRIAD-SILK, on the one hand, and standalone RocksDB and TRIAD, on the other hand, remain similar. Using compression is equivalent to working with a smaller dataset. Commit logging takes the same amount of bandwidth in all systems. Therefore, from an experimental perspective, using C_{log} is roughly equivalent to working on a machine with smaller disk bandwidth.

6.2 Nutanix workload

Workload description. We sample one of our production workloads at Nutanix over 24h. It is a write-dominated workload, with a 57:41:2 write:read:scan ratio (a scan length is in the order of tens of keys). The client requests arrive in bursts (peaks of around 20K requests/s), separated by periods of low activity (valleys of around hundreds of requests/s or less). A typical duration of a valley is between 5s and 20s, with an average valley length being approximately 15s. Most peaks (approximately 90%) are short bursts between 10s and 20s. The longer peaks (>100s) make up the rest of the workload. The maximum peak lasts approximately 400s. The request sizes range between 250B and 1KB for

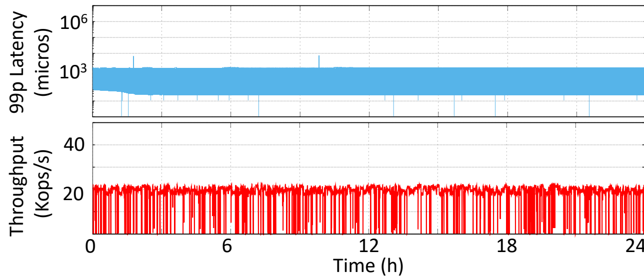


Figure 10: SILK in 24h Nutanix production workload. Top: 99th percentile latency, log scale on y-axis. Bottom: throughput. SILK maintains stable low latency and throughput close to the client load for extended time spans.

the single-point operations (i.e., reads and writes), with a median of 400B. We use a trace replay of the original workload, providing the input at the same rate as the original trace.

Results. Figure 8A shows the 99th percentile latency (left) and throughput (right) for RocksDB-SILK (bottom row), compared to state-of-the-art systems. Results obtained with TRIAD-SILK are similar.

SILK obtains two orders of magnitude lower tail latency than the auto-tuned RocksDB, and three orders of magnitude better than RocksDB and TRIAD, due to its combination of adjusting the I/O bandwidth and better internal work scheduling. Similar to their behavior described in Section 4, the tail latencies in RocksDB and TRIAD exhibit frequent spikes, due to stalling and contention for I/O bandwidth. The auto-tuned rate limiter in RocksDB achieves one order of magnitude better tail latency than both TRIAD and RocksDB, but does not avoid interference on shared resources as effectively as SILK (see Figure 8A, third row). The auto-tuner simply increases I/O bandwidth when there is more internal work to do, and it is oblivious of user load.

Throughput in SILK stays close to the offered client load, while throughput in RocksDB presents high fluctuations. Client operations build up in the worker thread queues because of interference with internal operations. When they can proceed, they are processed in bursts, which results in throughput spikes. TRIAD and the auto-tuned RocksDB stay closer to the offered client load, but still present throughput fluctuations, correlated to increases in tail latency.

Figure 9 shows the number of times C_m cannot be flushed right away (left) and the amount of time the writes are stalled, relative to the duration of the experiment (right). The statistics are collected for the experiment shown in Figure 8A. SILK never stalls writes and can always flush C_m as soon as it fills up. TRIAD has the most problems flushing C_m on time – the flush is delayed 69 times – because of its $L_0 - L_1$compaction strategy. The auto-tuned version of RocksDB does better, but it still spends a significant amount of the time stalling writes, consisting of 1% of the total experiment time.

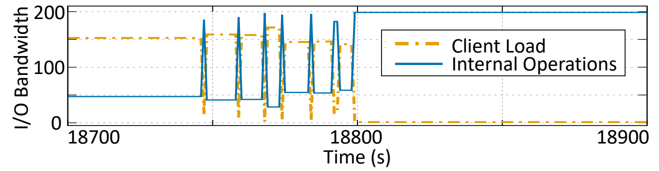


Figure 11: Detail of RocksDB-SILK I/O bandwidth allocation. SILK boosts internal work when client load decreases.

24h production workload. Figure 10 presents the 99th percentile latency and throughput time series of RocksDB-SILK for a 24h run of our production workload. SILK maintains stable performance over the extended period of time. Figure 11 shows a detail of the I/O bandwidth allocation in RocksDB-SILK during 200s of the production workload. Internal work may be temporarily postponed, but is eventually completed in the long term. RocksDB-SILK compacts approximately 3TB of data over the 24h and never has more than three compaction operations waiting to be scheduled. The worker threads experience no write stalls and have less than three operations enqueued throughout the experiment.

6.3 YCSB benchmarks

To evaluate the performance of SILK for a wide range of workloads, we now present results with the the full YCSB benchmark. From this point forward, we show results obtained with TRIAD-SILK. We report that the results for RocksDB-SILK are similar.

Workload description. YCSB provides six core workloads, described in Table 1. We use 8B keys and 1024B values. We evaluate SILK in the zipfian and uniform key distributions and show that SILK reduces tail latency in write-dominated workloads without inducing significant performance degradation in other scenarios.

Results. Figure 12 shows the average throughput of TRIAD and TRIAD-SILK, for the uniform (top) and zipfian (bottom) key distributions. SILK has low impact on throughput for both key distributions, amounting to at most 7%. As expected, SILK incurs the highest overhead for the uniform key distribution for the write-dominated workloads (i.e., YCSB F), because it entails frequent compactions and therefore frequent scheduling interventions by SILK. Reads do not suffer significantly compared to the baseline because of L0 read optimizations in TRIAD (and RocksDB): (1) there is a Bloom filter for each L0 file, and (2) reads return after the KV tuple is first found on L0, without checking all the L0 files. Because of these two optimizations, most of the time only one L0 file is read. So, even if SILK postpones L0 – L1 compactions, it has little impact on read performance. With a zipfian key distribution, most requests are served from memory in the entire benchmark, leading to less compaction. Here, SILK's overhead is at most 4%. Similarly, read-dominated workloads (YCSB B, C, D and E) are less impacted by com-

Workload	Description
YCSB A	write-intensive: 50% updates, 50% reads
YCSB B	read-intensive: 5% updates, 95% reads
YCSB C	read-only: 100% reads
YCSB D	read-latest: 5% updates, 95% reads
YCSB E	scan-intensive: 5% updates, 95% scans; average scan length 50 elements
YCSB F	50% read-modify-write, 50% reads

Table 1: YCSB core workloads description.

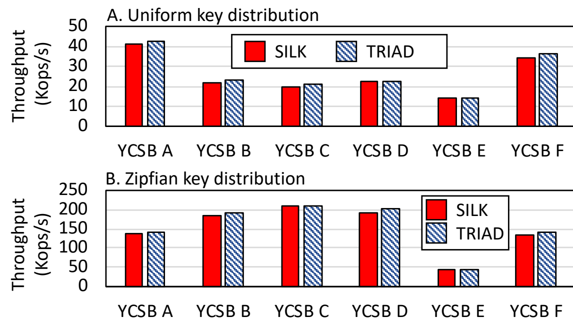


Figure 12: Average throughput of TRIAD and TRIAD-SILK in YCSB. Using SILK has minimal impact on throughput in read- and write- dominated workloads.

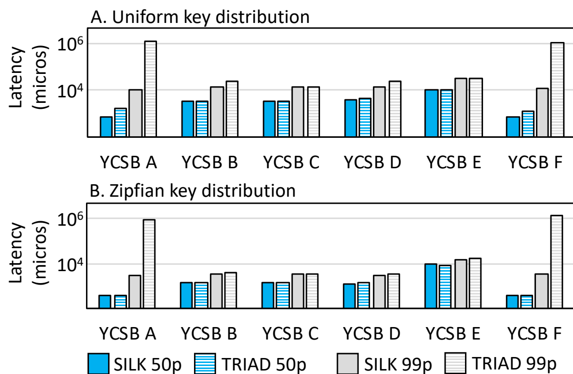


Figure 13: Latency of TRIAD and TRIAD-SILK in YCSB. Log-scale on Y-axis. SILK decreases 99p latency by two orders of magnitude in write-dominated workloads, while maintaining similar median latency across all workloads.

pactions, leading to less overhead (at most 5% in the uniform key distribution).

Figure 13 shows the median and 99 percentile latency for TRIAD and TRIAD-SILK. Generally, SILK's median latency is on par with that of TRIAD, or slightly lower. The only workload where SILK experiences higher median latency than TRIAD is YCSB E with a zipfian key distribution, where SILK surpasses TRIAD by 5%. Tail latency is lower with SILK across all workloads, by at least 5% (in YCSB E). SILK's benefits in terms of tail latency are most pronounced in write-dominated workloads, where the latency is decreased by up to two orders of magnitude.

6.4 Stress testing for long peaks

Workload description. In this section we focus on workloads in the style of YCSB core workload A (see Table 1), where we vary the ratio between the length of the client load peaks and valleys (gradually increasing peak duration, while keeping valley duration constant). We use 8B keys, 1024B values and a uniform key distribution. Client load during low activity periods is approximately 10 Koperations/second and approximately 40 Koperations/s during peaks. The offered load is higher for both the peaks and the valleys than our production workload, in order to stress the system.

Results. Figure 8B shows the 99th percentile latency (left) and the throughput (right) of TRIAD-SILK. The first three rows show a 50:50 write:read workload, where the ratio of peak:valley length is varied: peaks last 10, 50, and 100 seconds, while the valleys last 10 seconds. SILK easily sustains these peaks and valleys in the client load, keeping tail latency low and the throughput steady.

On the last two rows of the figure we show the results of an experiment with a long peak, to see at what point SILK's performance starts to degrade. The fourth row shows the results with a 50:50 write:read workload and the last row with a 90:10 write:read workload. Despite the prioritization of critical internal work, if the peak load is high and the peak duration long, the system cannot allocate enough resources to the internal work, and the performance eventually starts to degrade. Also, as expected, the proportion of writes influences the amount of time the peaks can be sustained. SILK's performance starts to degrade at around 500 seconds (300 seconds of peak) for the 90% writes workload, while the peak can be sustained up to around 700 seconds (500 seconds of peak) for the 50% writes scenario. Despite showing performance degradation, SILK is able to handle challenging workloads that are representative of real applications. Our production workload has at most 400s peaks, 50% writes and load peaks reaching only half the load of the synthetic workload peaks (Figure 10).

6.5 Breakdown

Figure 8C shows 99th percentile latency (left) and throughput (right) for the following variants of SILK. The first row shows a version where we enable SILK's dynamic I/O bandwidth rate limiting, but where the priorities and preemption are disabled. The second row shows the complementary version which uses priorities and preemption but where the I/O bandwidth is not controlled. The final row shows SILK. On their own, neither of the two techniques is able to sustain the client load.

In the first case (top row), the dynamic bandwidth allocation ensures that internal and external work interference is low. However, as the experiment progresses and larger compactions need to take place, the urgent internal operations are slowed down.

In the second case (middle row), good prioritization maintains the tree structure at the levels close to the memory component, allowing flushes and $L_0 - L_1$ compactions to proceed without slowdowns. However, as larger compactions need to take place, the fact that the bandwidth is not controlled leads to negative interference between internal and external work.

7 Related work

Reducing compaction overhead. Many systems reduce the overhead of compaction algorithms used in production systems, such as RocksDB [18], LevelDB [14] and Cassandra [30]. WiscKey [32], HashKV [10] and LWC-tree [44] separate keys from values, and only store keys in the LSM tree, reducing data movement in compaction operations. TRIAD [4] keeps hot data in memory, avoids duplicate writes of the log component, and compacts an SSTable only when there is sufficient overlap with lower-level SSTables. SlimDB [40] allows overlapping key-ranges on each level of the LSM tree to reduce the amount of data that is rewritten, and uses new index blocks and cuckoo filters to perform fast key lookup. Monkey [12], Dostoevsky [13], Lim et al. [31], Dong et al. [17] tune the parameters of the LSM tree, in order to limit the amount of maintenance work and to achieve better performance. Accordion [8] optimizes the layout of the in-memory component by means of a hierarchical structure and in-memory compactions. SifrDB [33] employs different compaction algorithms on different levels of the LSM tree.

These techniques decrease the amount of work performed during internal operations, with the result of increasing throughput. However, they do not avoid the interference with user operations *while* internal operations execute. By contrast, SILK schedules internal operations so as to avoid interference on user operations, thus avoiding latency spikes for user operations and improving tail latencies. The techniques of SILK can be applied to existing designs, thereby preserving their improvements in terms of internal operation overhead. We show this by applying SILK techniques on top of two systems: RocksDB and TRIAD.

Compaction variants and alternatives. PebblesDB [39] allows overlapping key ranges in the lower LSM tree levels to avoid SSTable merges, and uses a skip-list-like structure to allow efficient key lookups. PebblesDB achieves remarkable performance, but its last-level compaction may lead to prolonged service unavailability (Section 4). SILK techniques can be applied to PebblesDB to improve its robustness.

Tucana [38] and ForestDB [2] use variants of the B-ε tree [6, 9, 27] and of the B+ tree, respectively. Unlike LSM trees, these systems do not maintain large sorted files and hence do not implement flushing and compaction. However, they implement operations such as leaf splitting, leaf merging and tree re-balancing to preserve the structure of the tree. These operations result in random accesses that increase write amplification and affect the latency of user operations by contending for I/O. SILK targets LSM tree-based systems, which favor sequential I/O, absorb writes in memory, and leverage sorted SSTables for efficient range scans.

Ahmad and Kemme [1] offload compaction to a dedicated server. Atlas [29] uses different servers to store keys and values. Using a different server for compactions is an effective way to address latency spikes, since this approach removes interference between client and internal operations. However, this solution substantially changes the architecture of the KV store from a standalone system to a distributed one, which results in higher operational costs and increased complexity. Moreover, the transfer of SSTables between the compaction server and the client-facing servers puts additional load on the network, which can generate interference on co-located applications.

Data structure and algorithm improvements. FloDB [5], cLSM [24], HyperLevelDB [26], Nibble [35] and BespoKV [3] improve scalability by alleviating contention bottlenecks. NoveLSM [28] reduces logging overhead by supporting in-place updates to a component stored on NVM, and performs parallel reads. These techniques are orthogonal to SILK's. Minos [16] reduces tail latency for in-memory KVs focusing on workloads with heterogeneous item sizes. To this end, Minos serves similar-sized requests on the same cores. Similar approaches could be implemented in LSM KVs to further reduce tail latency in heterogeneous workloads, and they can co-exist with SILK. bLSM [41] aims to avoid stalling at a level L of the tree by ensuring that operations at lower levels have completed by the time level L has to push data to lower levels. bLSM achieves this goal by throttling internal operation rates. bLSM, however, may throttle user writes as the memory component fills up. Instead of artificially throttling requests, SILK performs internal operations during off-peak periods, and prioritizes higher-level internal operations to avoid stalling user operations.

8 Conclusion

In this paper we presented SILK, a new LSM KV store designed to prevent client request latency spikes. SILK uses an I/O scheduler to manage external client load and internal LSM maintenance work. We implemented SILK in two state-of-the-art LSM KVs and demonstrated order-of-magnitude improvements in latency at the 99th percentile in synthetic and production workloads from Nutanix.

Acknowledgements. We would like to thank our shepherd, Vijay Chidambaram, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Swiss National Science Foundation through grant No. 513954, an EcoCloud Postdoctoral Fellowship, and by a gift from Nutanix, Inc. Part of the work has been done while Oana Balmau was an intern at Nutanix.

References

[1] AHMAD, M. Y., AND KEMME, B. Compaction Management in Distributed Key-value Datastores. In *Proceedings of VLDB* (2015).

[2] AHN, J., SEO, C., MAYURAM, R., YASEEN, R., KIM, J., AND MAENG, S. ForestDB: A Fast Key-value Storage System for Variable-length String Keys. *IEEE Transactions on Computers 65*, 3.

[3] ANWAR, A., CHENG, Y., HUANG, H., HAN, J., SIM, H., LEE, D., DOUGLIS, F., AND BUTT, A. R. BespoKV: Application Tailored Scale-out Key-value Stores. In *Proceedings of SC18* (2018).

[4] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-value Stores. In *Proceedings of USENIX ATC* (2017).

[5] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. FloDB: Unlocking Memory in Persistent Key-value Stores. In *Proceedings of EuroSys* (2017).

[6] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An Introduction to Bε-trees and Write-optimization. *;login: 40*, 5 (2015).

[7] BLOOM, B. H. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970).

[8] BORTNIKOV, E., BRAGINSKY, A., HILLEL, E., KEIDAR, I., AND SHEFFI, G. Accordion: Better Memory Organization for LSM Key-value Stores. In *Proceedings of VLDB* (2018).

[9] BRODAL, G. S., AND FAGERBERG, R. Lower Bounds for External Memory Dictionaries. In *Proceedings of SODA* (2003).

[10] CHAN, H. H. W., LI, Y., LEE, P. P. C., AND XU, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of USENIX ATC* (2018).

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC* (2010).

[12] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal Navigable Key-value Store. In *Proceedings of SIGMOD* (2017).

[13] DAYAN, N., AND IDREOS, S. Dostoevsky: Better Space-time Trade-offs for LSM-tree Based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of SIGMOD* (2018).

[14] DEAN, J., AND GHEMAWAT, S. LevelDB. `https://github.com/google/leveldb`. visited Jan 2019.

[15] DELIMITROU, C., AND KOZYRAKIS, C. Amdahl's Law for Tail Latency. *Communications of the ACM 61*, 8 (2018).

[16] DIDONA, D., AND ZWAENEPOEL, W. Size-aware Sharding for Improving Tail Latencies in In-memory Key-value Stores. In *Proceedings of NSDI* (2019).

[17] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing Space Amplification in RocksDB. In *Proceedings of CIDR* (2017).

[18] FACEBOOK. RocksDB: A Persistent Key-value Store for Fast Storage Environments. `https://rocksdb.org`. visited Jan 2019.

[19] FACEBOOK. RocksDB Autotuned Rate Limiter. `https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html`. visited Jan 2019.

[20] FACEBOOK. RocksDB Benchmarking Tools. `https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`. visited Jan 2019.

[21] FACEBOOK. RocksDB Level-based Compaction Changes. `https://rocksdb.org/blog/2017/06/26/17-level-based-changes.html`. visited Jan 2019.

[22] FACEBOOK. RocksDB Rate Limiter. `https://github.com/facebook/rocksdb/wiki/Rate-Limiter`. visited Jan 2019.

[23] FACEBOOK. RocksDB Tuing Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`. visited Jan 2019.

[24] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling Concurrent Log-structured Data Stores. In *Proceedings of EuroSys* (2015).

[25] HUA, Y., XIAO, B., VEERAVALLI, B., AND FENG, D. Locality-sensitive Bloom Filter for Approximate Membership Query. *IEEE Transactions on Computers 61*, 6 (2012).

[26] HYPERDEX. HyperLevelDB. `https://github.com/rescrv/HyperLevelDB`. visited Jan 2019.

[27] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: Write-optimization in a Kernel File System. *ACM Transactions on Storage (TOS) 11*, 4 (2015).

[28] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of USENIX ATC* (2018).

[29] LAI, C., JIANG, S., YANG, L., LIN, S., SUN, G., HOU, Z., CUI, C., AND CONG, J. Atlas: Baidu's Key-value Storage System for Cloud Data. In *Proceedings of MSST* (2015).

[30] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010).

[31] LIM, H., ANDERSEN, D. G., AND KAMINSKY, M. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. In *Proceedings of FAST* (2016).

[32] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of FAST* (2016).

[33] MEI, F., CAO, Q., JIANG, H., AND LI, J. SifrDB: A Unified Solution for Write-optimized Key-value Stores in Large Datacenter. In *Proceedings of SoCC* (2018).

[34] MEI, F., CAO, Q., JIANG, H., AND TINTRI, L. T. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of SoCC* (2017).

[35] MERRITT, A., GAVRILOVSKA, A., CHEN, Y., AND MILOJICIC, D. Concurrent Log-structured Memory for Many-core Key-value Stores. In *Proceedings of VLDB* (2017).

[36] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-Structured Merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996).

[37] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O Bottleneck: A Case for Log-structured File Systems. *ACM SIGOPS Operating Systems Review 23*, 1 (1989).

[38] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proceedings of USENIX ATC* (2016).

[39] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of SOSP* (2017).

[40] REN, K., ZHENG, Q., ARULRAJ, J., AND GIBSON, G. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. In *Proceedings of VLDB* (2017).

[41] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of SIGMOD* (2012).

[42] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of EuroSys* (2014).

[43] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of USENIX ATC* (2015).

[44] YAO, T., WAN, J., HUANG, P., HE, X., WU, F., AND XIE, C. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage (TOS) 13*, 4 (2017).

Unification of Temporary Storage in the NodeKernel Architecture

Patrick Stuedi[†] Animesh Trivedi[‡] Jonas Pfefferle[†] Ana Klimovic[§]
Adrian Schuepbach[†] Bernard Metzler[†]
[†]*IBM Research* [‡]*Vrije Universiteit* [§]*Stanford University*

Abstract

Efficiently exchanging temporary data between tasks is critical to the end-to-end performance of many data processing frameworks and applications. Unfortunately, the diverse nature of temporary data creates storage demands that often fall between the sweet spots of traditional storage platforms, such as file systems or key-value stores.

We present NodeKernel, a novel distributed storage architecture that offers a convenient new point in the design space by fusing file system and key-value semantics in a common storage kernel while leveraging modern networking and storage hardware to achieve high performance and cost-efficiency. NodeKernel provides hierarchical naming, high scalability, and close to bare-metal performance for a wide range of data sizes and access patterns that are characteristic of temporary data. We show that storing temporary data in Crail, our concrete implementation of the NodeKernel architecture which uses RDMA networking with tiered DRAM/NVMe-Flash storage, improves NoSQL workload performance by up to 4.8× and Spark application performance by up to 3.4×. Furthermore, by storing data across NVMe Flash and DRAM storage tiers, Crail reduces storage cost by up to 8× compared to DRAM-only storage systems.

1 Introduction

Managing temporary data efficiently is key to the performance of cluster computing workloads. For example, application frameworks often cache input data or share intermediate data, both both within a job (e.g., shuffle data in a map-reduce job) and between jobs (e.g., pre-processed images in a machine learning training workflow). Temporary data storage is also increasingly important in serverless computing for exchanging data between different stages of tasks [17].

Storing temporary data efficiently is challenging as its characteristics typically lie between the design points of existing storage platforms, such as distributed file systems and key-value stores. For instance, shuffle data in a map-reduce job may consist of a large number of files which are organized hierarchically, vary widely in size, are written randomly, and read sequentially. While file systems (e.g., HDFS) offer a convenient hierarchical namespace and efficiently store large datasets for sequential access, distributed key-value stores are optimized for scalable access to a large number of small objects. Similarly, DRAM-based key-value stores (e.g., Redis) offer the required low latency, but persistent storage platforms (e.g, S3) are more suitable for high capacity at low cost. Overall, we find that existing storage platforms are not able to satisfy all the diverse requirements for temporary data storage and sharing in distributed data processing workloads.

In this paper we present *NodeKernel*, a new distributed storage architecture designed from the ground up to support fast and efficient storage of temporary data. As its most distinguishing property, the NodeKernel architecture fuses file system and key-value semantics while leveraging modern networking and storage hardware to achieve high performance. NodeKernel is based on two key observations. First, many features offered by long-term storage platforms, such as durability and fault-tolerance, are not critical when storing temporary data. We observe that under such circumstances, the software architectures of file systems and key-value stores begin to look surprisingly similar. The fundamental difference is that file systems require an extra level of indirection to map offsets in file streams to distributed storage resources, while key-value stores map entire key-value pairs to storage resources. The second observation is that low-latency networking hardware and multi-CPU many-core servers dramatically reduce the cost of this indirection in a distributed setting by enabling scalable RPC communication at latencies of a few microseconds.

Based on these insights we develop the NodeKernel architecture by implementing file system and key-value semantics as thin layers on top of a common storage kernel. The storage kernel operates on opaque data objects called "nodes" in a unified namespace. Applications can store arbitrary size data in nodes, arrange nodes in a hierarchical namespace, and obtain file system or key-value semantics through specialized

node types if needed. For instance, key-value and table nodes permit concurrent creation of nodes with the same name, offering last-put-wins semantics. On the other hand, file and directory nodes permit efficient enumeration of data sets at a given level in the storage hierarchy. By splitting functionality in a common storage kernel and custom node types, NodeKernel enables applications to use a single platform to store data that may require different semantics, while generally offering good performance for a wide range of data sizes and access patterns.

NodeKernel is designed explicitly with modern hardware in mind. Following strict separation of concerns, the storage kernel is composed of a lightweight and scalable metadata plane tailored to low-latency networking hardware and an efficient data plane that provides access to multiple tiers of network-attached storage resources. The metadata plane is trimmed down to offer only the most critical functionality, with low overhead. The data plane runs a lightweight software stack and leverages modern networking and storage hardware to achieve fast access to arbitrary size data sets while also optimizing cost efficiency. For instance, data attached to "nodes" may either be pinned to a particular storage technology tier or may spill from one storage tier to another depending on performance and cost requirements.

Crail is our concrete implementation of the NodeKernel architecture using RDMA networking and two storage tiers based on DRAM and NVMe SSDs respectively. We evaluated Crail on a 100Gb/s RoCE cluster equipped with Intel Optane NVMe SSDs using raw storage microbenchmarks as well as using the NoSQL YCSB benchmark and different Spark workloads. Our results show that Crail matches the performance of current state-of-the-art file systems and key-value stores when operated in their sweet spot, and outperforms existing systems up to 3.4× for data accesses outside the sweet spot of file systems and key-value stores. Moreover, Crail creates new opportunities to reduce cost and gain flexibility with almost no performance penalties by using NVMe Flash in addition to DRAM. For instance, using Crail to store shuffle data in Spark allows us to adjust the ratio between DRAM and Flash with only a minimal increase in job runtimes.

In summary, this paper makes the following contributions:

- We propose NodeKernel, a new storage architecture fusing file system and key-value semantics to best meet the needs of temporary data storage in data processing workloads.

- We present Crail, a concrete implementation of the NodeKernel architecture using RDMA, DRAM, and NVMe Flash.

- We show that storing temporary data in Crail reduces the runtime and cost of data processing workloads. For instance, Crail improves performance up to 4.8× for NoSQL workloads. When integrated in Spark's shuffle and broadcast services, Crail improves application performance up to 3.4× and reduces cost up to 8×.

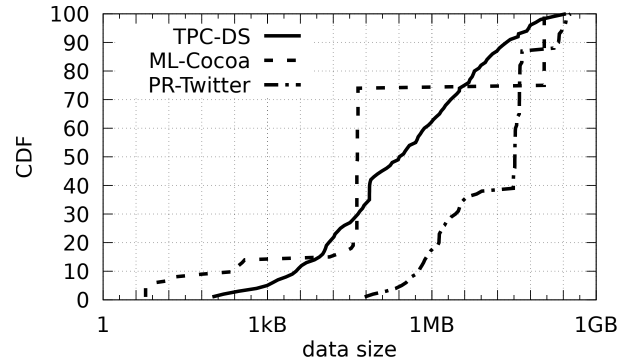


Figure 1: CDF of the size of intermediate data written or read per compute task in Spark for different workloads.

Crail is an open source Apache project [2, 3] with the code available for download from the project website as well as directly from GitHub at https://github.com/apache/incubator-crail. Further, all benchmarks used in this paper are open source.

2 Background and Motivation

Temporary data represent a large and important class of in-processing data in analytics frameworks. For example, Zhang et al. report that over 50% Spark jobs executed at Facebook contain at least one shuffle operation, generating significant amounts of temporary data [37].

We define temporary data as the multitude of all application data being created, handled, or consumed during processing, excluding the original input and final output data. Specifically, we identify three distinct classes of temporary data: intra-job, inter-job, and cached input/output datasets. Intra-job temporary data is generated within a framework when executing a single job like page-rank, or a SQL query. Common examples are datasets generated during shuffle or broadcast operations in frameworks like Spark, Hadoop or Flink. Such data is typically generated and consumed by the same job, which puts a bound on the lifetime of the data. Inter-job temporary data are intermediate results in multi-job pipelines. For example, there are many pre-processing and post-processing jobs in a typical machine learning pipeline [35], where the output of one job becomes the input of another job. Lastly, examples of cached input/output data are mostly read-only datasets that are pulled into a cache for fast repetitive processing. For instance, users may run many SQL queries on the same table (or a view) over a short period of time. In this case, a copy of the input table might be cached on a fast storage media.

Technology	Local Latency	Remote Latency	Bandwidth	Price
DRAM	80ns	2us	10s GB/s	5$/GB
3D XPoint	5us	10us	2-3 GB/s	1.25$/GB
NAND Flash	50us	55us	2-3 GB/s	0.63$/GB

Table 1: Price and performance of DRAM (DDR4), 3D XPoint (NVMe) and NAND Flash (NVMe). Remote DRAM latency for RDMA, remote 3D XPoint and NAND Flash latency for NVMf.

Building an efficient storage platform for the different types of temporary data requires careful consideration of application demands, data characteristics and hardware opportunities. In the following section we discuss several requirements for a temporary storage platform and provide an overview of current state-of-the-art solutions.

2.1 Requirements and Challenges

Size, API, and Abstractions Diversity: Temporary data in data processing workloads can vary substantially with regard to the data size. In Figure 1 we show the size distribution (CDF) of temporary data generated per task during the execution of (a) PageRank on the Twitter graph; (b) SQL queries on a TPC-DS dataset; and (c) Cocoa machine learning on a sparse matrix dataset [24]. As shown, the per-task data sizes are ranging from a few bytes (for machine learning) to a GB (for TPC-DS). Historically, different storage systems are used to handle the two ends of this spectrum. Distributed key-value stores (e.g., RAMCloud, memcached, etc.) have an object API and are optimized to store small values efficiently for fast random lookups [20, 33]. In contrast, file systems like HDFS or Ceph, can store large datasets (GBs) efficiently by partitioning the dataset and maintaining indexes for lookups. Moreover, filesystem abstractions of appendable files, enumerable hierarchical namespace, and a streaming-byte interface for I/O provide additional support for an easy mapping of temporary datasets, such as all-to-all shuffle, to the underlying storage.

A temporary storage platform should be able to store small and large values efficiently, with the unified benefit of file and key-value abstractions in a single system.

Performance: Temporary data often lies in the critical path of data processing, hence it is imperative that access to the temporary data is fast. As with the size, the access pattern also varies widely. For example, as there is no global order, shuffle data is often written randomly [11], whereas SQL tables are read in large sequential scans [32]. Hence, one requirement a storage platform for temporary data has to fulfill is that it should be capable of performing well on the entire spectrum of data sizes for any access pattern.

Fortunately, over the last decade, I/O devices have evolved rapidly to support high-bandwidth (100s of Gbps), ultra low-latencies (less than 10 usec), with millions of IOPS. In order to meet data processing demands, these devices are now being deployed in the cloud (AWS, Azure), and used inside data processing frameworks. Consequently, an ideal temporary storage system should be able to run efficiently on modern networking and storage hardware while delivering close to bare-metal performance.

Beyond In-Memory Storage: The total amount of temporary data that is generated or consumed by data processing workloads can be large. For instance, between the workloads whose temporary data object size CDFs are shown in Figure 1, the collective total volume of data differs by 100s of GBs (10-100% of the input dataset size, not shown in the figure). Efficiently storing large volumes of data while offering good data access performance is difficult. For instance, storing all the data in DRAM is preferred from a performance standpoint but doing so typically is too costly. Thankfully, over the past years different media types such as NAND Flash, and PCM storage, have emerged to store data at a different cost, performance, and energy price point. Hence, an efficient storage platform for temporary data should integrate multiple storage technologies that offer different performance cost trade-offs, and allow applications to choose between different points in the trade-off space. A comparison of different storage technologies with respect to price and performance is given in Table 1.

Non-Requirements: We observe that in the specific case of temporary data storage, many traditional storage features such as durability and fault-tolerance are not a priority. Durability, for instance, is of low importance due to the short lifetime of temporary data. While fault-tolerance is generally useful for short-lived data, it still is not a high priority for temporary data storage. Today, fault tolerance is often implemented at the level of the compute framework, in a coarse grained manner. For instance, Spark [36] and Ray [25] use lineage tracking to re-compute data in case of data loss by relaunching tasks.

2.2 Limitations of Existing Approaches

We review current state-of-the-art storage systems with regard to the design goals listed in the previous section. We classify the systems discussed into the following three categories.

Key-Value Stores: Memcached [4] and Redis [5] are two of the most popular key-value stores designed to store data in DRAM. Network-optimized KVs like MICA [21], Herd [14], FaRM [12], KVDirect [19] and RAMCloud [26] use RDMA operations to provide high-performance data accesses but cannot easily integrate data storage to different tiers beyond DRAM. Redis has an extension to spill data to Flash, however keys are still stored in DRAM, and hence are limited by the DRAM capacity. Storage-optimized KVs such as Aerospike [30] or BlueCache [34], use NAND Flash for storage. Hence, their performance is bounded by the performance of Flash, and they are not optimized for the next-generation

of NVM storage devices like Optane (see Section 5.1). Other systems, like HiKV [33], have hybrid DRAM-Flash indexes but only target a single node deployment, hence limiting their applications to a wider class of operations such as shuffling. Furthermore, the design of these KV stores is tailored to very small data sets of a few hundred bytes up to a few MB maximum, thus, limiting their operational window to these object sizes.

Distributed Data Stores: Storage systems such as Cephover-Accelio [8] and Octopus [23] are high-performance distributed file systems used for fast network and NVM devices. However, due to the focus on providing fault-tolerant, durable storage their performance for small objects is poor (see Section 5). The recently proposed Regions system provides a file abstraction to remote memory [6]. As with other in-memory storage systems, however, Regions is not a cost-effective solution for storing large data sets. Systems like Alluxio [1] and Pocket [17] provide support for multiple storage technology types. But Alluxio does not deliver the performance of high-end hardware, and is targeted towards building local caches. Pocket shares our aim of a dedicated storage system for temporary data and its design has similarities with the NodeKernel architecture. However, the focus of Pocket is on providing efficient and elastic temporary storage on commodity hardware in the cloud whereas NodeKernel is designed for low-latency high-bandwidth network and storage hardware. Moreover, Pocket has only an object based I/O interface which is well suited for data sharing in serverless workloads. In contrast, NodeKernel's unified API provides semantics like "append" and "bag" to support storing a wide range of temporary data in different workloads.

Temporary-data specific operations: A number of works accelerate specific storage operations in data processing workloads. For instance, Riffle [37] is an optimized shuffle server that aims to reduce overheads associated with large fanouts. Sailfish [27] is a framework that introduced I-files which are shuffle optimized data containers. ThemisMR [28] also aims to optimize shuffle and target small rack-scale deployments. In general, the aim of these systems is to optimize disk-based, file-oriented shuffle data management for map-reduce type workloads. It is not clear how their design can support other communication patterns, such as broadcast and multicast, or integrate different storage types to optimize for different access patterns. Parallel databases [7, 22] use RDMA-optimized shuffling operations for database operators. These works, however, are highly database specific and do not extend naturally to other data processing workloads or other forms of temporary data.

3 The NodeKernel Architecture

We present the NodeKernel, a new storage architecture designed to match the diverse and complex demands of temporary data storage in data processing workloads. The NodeKernel tackles this challenge by fusing storage semantics that are otherwise available separately in file systems and key-value stores, such as hierarchical naming, scalability, multiple storage tiers, fast enumeration of datasets, and support for both tiny and large data sizes (Figure 2). The NodeKernel architecture is guided by three design principles:

1. Distill higher-level storage semantics into thin layers on top of a common storage kernel.
2. Separate data management concerns into a lightweight metadata plane and a "dumb" data plane optimized for modern networking and storage hardware.
3. Leverage multiple storage technologies for efficient storage of large datasets.

We discuss each of these design principles in more detail below before describing Crail, a concrete implementation of the NodeKernel architecture, in Section 4.

3.1 Storage Kernel and Node Types

In the NodeKernel architecture, higher-level storage semantics are implemented as thin layers – or more precisely, as specialized data types – on top of a shared storage kernel exporting a hierarchical namespace of opaque data "nodes". Nodes are objects of an abstract type `Node` as shown in the following code snippet.

```
abstract class Node {
protected:
  /*implemented by derived types*/
  abstract bool addChild(Node child);
  abstract bool removeChild(Node child);
  /*implemented by the storage kernel*/
  future<int> read(byte[] buf);
  future<int> append(byte[] buf);
  future<int> update(byte[] buf, int off);
  string getPath();
  int size();
  ...
}
```

The kernel is responsible for allocating storage resources on behalf of nodes, manipulating the hierarchical namespace, and implementing basic data access operations such as `read`, `append` and `update`. Applications interface with the storage kernel to create data nodes at a given location in the hierarchy, attach data of arbitrary size to a node, look up nodes, and fetch the associated dataset from a node. Nodes are identified using path names encoding the location in the storage hierarchy, similar to files and directories in a file system.

Applications do not create raw `Node` objects directly, instead they create objects of derived types offering specialized functionality. These so-called custom types implement higher-level storage semantics by extending `Node` in two ways. First,

custom types provide implementations for the abstract operations addChild and removeChild. These operations are called by the kernel whenever a new node is inserted or removed to/from the storage hierarchy. Second, custom data types provide specialized data access operations implemented using read and append available in Node.

NodeKernel defines five custom node types, each offering slightly different semantics and operations:

- File and KeyValue: Both node types provide read and append interfaces by exposing the corresponding operations in Node. The two types, however, provide different semantics during the creation and insertion of new nodes, controlled via the implementation of addChild and removeChild. For File nodes, the first create operation for a given path name succeeds and subsequent create operations on the same path name fail. For KeyValue nodes, subsequent create operations on a path name representing an existing node will succeed, replacing the existing node. As we will see in Section 5, KeyValue nodes are useful to cache input datasets in NoSQL workloads, permitting concurrent updates of the data, whereas File nodes are a better match to cache read-only input data in Spark workloads.

- Directory and Table: Those node types are containers for File and KeyValue nodes respectively. Directory and Table nodes store the name components of all of their children as part of their data (implemented using append and update operations available in Node). For instance, the data segment of a Directory with path name "/a/b" storing two files with path names "/a/b/c1" and "/a/b/c2" consists of the two name components "c1" and "c2". Both Directory and Table nodes offer operations to enumerate the names of all of their children (implemented using read available in Node). Tables can optionally be configured as "non-enumerable" in which case no name components are stored and enumeration returns the empty set. Creating KeyValue nodes in a non-enumerable table is typically faster because it eliminates the step of updating the name component (as described in Section 4.1).

- Bag: The Bag node type is designed to support efficient sequential reading of data spread across many data nodes. A Bag behaves like a directory such that it acts as a container for File nodes. Applications create and write files in a Bag just like they create and write files in a Directory. When reading a Bag, however, the Bag appears to the application like a single file. Using a Bag's read operation allows applications to sequentially read through the full set of files in the bag. Generally, the read operation of a Bag offers better performance than reading each File node separately, due to more efficient metadata access at file boundaries. As we will see in Section 5, Spark applications can use Bags to store shuffle data, allowing reduce tasks to efficiently fetch data written by map tasks. The Bag type in NodeKernel is similar in spirit than bags in Hurricane, a recent work on taming skew in data processing workloads [9].

NodeKernel restricts the way node types can be stacked. For instance, KeyValue nodes can only be attached to Table nodes, whereas File nodes can only be attached to Bag and Directory nodes. Moreover, directories can be arbitrarily nested, whereas bags and tables implement a flat namespace. The permitted combinations of node types match the specific use cases of temporary data storage. At the same time, preventing arbitrary combinations of node types ensures the architecture is not over-designed and simple to implement. For instance, Bags are mainly used to store shuffle data which is organized in flat per-reducer buckets, thus, enabling arbitrarily nested bags seemed unnecessary. At the time, a read operation on flat Bags is easier to implement than a read operation on an arbitrarily nested Bag.

Why provide a single unified storage namespace? By splitting functionality in a common storage kernel and a set of custom data types, the NodeKernel achieves two things. First, it permits different types of temporary data requiring different semantics to be managed by a single storage platform. For instance, as we will see later, Spark applications can use Crail for storing both broadcast and shuffle data, as well as for caching RDDs. Second, decoupling core data access from storage semantics allows applications to choose a particular node type based on the semantics they need (key/value vs file) rather than based on the size of the data or the access pattern. As discussed in Section 2.1, temporary data often varies in terms data size and access pattern even within a single workload, making it difficult to store the data efficiently in a storage platform like a filesystem or a key-value store. By contrast, node types in NodeKernel have no size limitation and provide efficient data access for different access patterns as we will see later in Section 5.

3.2 System Architecture

Figure 2 illustrates the NodeKernel system architecture. At the data management level, NodeKernel's architecture resembles the architecture of distributed file systems like HDFS or GFS, consisting of a set of metadata and storage servers deployed across a cluster. Data attached to a "node" in the storage hierarchy appears to clients as a stream, but internally the data is composed of a sequence of blocks. A block refers to a fixed sequence of bytes stored in one of the storage servers. Metadata servers maintain the hierarchical storage namespace as well as block metadata, i.e., a mapping between storage blocks and storage servers. Storage servers allocate a large set of storage blocks at startup and register them with

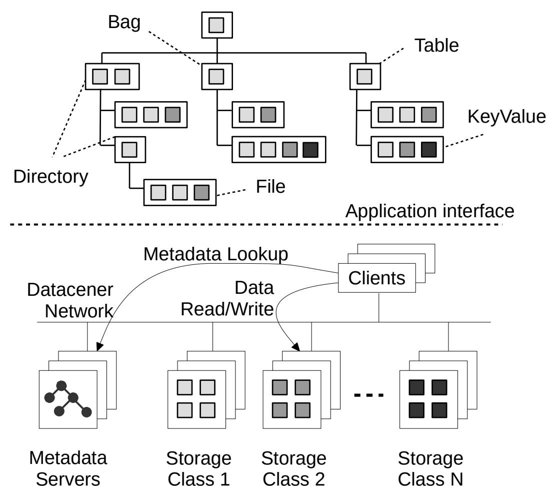


Figure 2: The NodeKernel storage architecture.

one of the metadata servers. Metadata servers maintain a free list with blocks that are not assigned to a particular node, and move blocks from the free list to a per node list during data writes and appends. When accessing data, clients first contact one of the metadata servers and request the metadata for the corresponding block. Based on this information clients then contact the given storage server to read or write the data.

The `Node` abstract data type exports two abstract operations, `addChild` and `removeChild`, to be implemented by the derived types described in Section 3.1. Those operations are executed at the metadata server each time a new node is created or removed. The `Node` further exports functions to manipulate data such as `read`, `append` or `update`. Those functions are implemented as part of the client library and require interactions with both metadata and storage servers. Finally, the basic metadata operations such as `getPath` or `size` are also implemented by the client library returning cached values if possible.

Target deployment: NodeKernel targets temporary data which is short-lived and simple to regenerate. Furthermore, NodeKernel targets small to medium size deployments consisting of few compute racks. Considering the target deployments and the nature of the data to be stored, NodeKernel prioritizes performance over fault tolerance. However, if deemed necessary, additional fault-tolerance mechanism such as replication, erasure-coding, etc., can be added. The lack of these features is not fundamental to the design.

Performance challenges: The main challenge in NodeKernel was to come up with a system architecture that can serve the full spectrum of demands discussed in Section 2.1. In particular, the architecture should be scalable like a key-value store despite offering a convenient hierarchical namespace. Furthermore, the architecture should support both low-latency key-value style access, as well as high-bandwidth file system like data access. In the following we discuss in how we accommodate these requirements in the NodeKernel architecture.

3.2.1 Low-latency metadata operations

Fusing file system and key-value semantics into a single storage kernel primarily requires a fast metadata access. In Section 2.1, we observed that software architectures of distributed file systems fundamentally differ from key-value stores only by an extra metadata operation required to map offsets in file streams to storage resources (e.g., blocks on storage servers). Thus, keeping the overheads of metadata operations low will make NodeKernel's architecture amenable to key-value style operations on small datasets, while also improving the efficiency of large data accesses.

Today, modern low-latency networking hardware enables RPC communication at latencies of a few microseconds [13, 16]. The NodeKernel features a lightweight metadata plane that matches well with low-latency networking hardware. For one, the metadata plane is trimmed down to offer only the most critical functionality consisting of six key RPC operations: `create` to create a new node, `lookup` to retrieve the metadata of a node, `remove` to delete an existing node, `move` to move a node to a different location in the storage hierarchy, `map` to map a logical offset in a node's data stream to a storage resource, and `register` used by storage servers to register storage resources with the metadata server. All these operations have low compute and I/O intensity and can be implemented at the latency of a few microseconds on a high-performance network fabric. We deliberately move data intensive metadata operations like `enumeration` to the data plane to avoid interference (see Section 3.1 and 4).

3.2.2 Metadata partitioning

The scalability of NodeKernel heavily depends on the throughput and scalability of its metadata RPC subsystem. Recent work has shown that RPC systems can be scaled to millions of operations per second on a single server using high-performance networking hardware on the one hand [15], but also using efficient software stacks on commodity hardware [13]. The lightweight RPC interface in NodeKernel is designed for high throughput and can drive up to 10 million metadata operations per second using a single metadata server, as shown later in Section 5. Up to now in all of our deployments we have never had a situation where a single metadata server would reach its limit. In fact, in one of our largest deployments on 128 nodes the metadata throughout reached 4.5 million operations per second, thus, roughly half of what a single metadata server can support.

Nevertheless, for the case where a single metadata server

is not sufficient, NodeKernel permits partitioning the metadata space over multiple servers. Thereby, the top-level root namespace is hash-partitioned among an ordered list of metadata servers. Using metadata partitioning, one can scale the metadata plane horizontally assuming a sufficiently large top level fan out.

One drawback of static of partitioning based on top-level namespaces is that load may be unevenly distributed among the metadata servers depending on the size of the subtree and the activity within the subtree. One alternative approach would be dynamic partitioning at a more fine grained level. For instance, past work has proposed a partitioning scheme for file systems where metadata partitioning is implemented at the directory level, with an option to split large directories on-demand as they grow too big and distribute the splits over multiple metadata servers [29]. Even though such an approach creates a more even load distribution, it comes at a significant performance cost as it requires multiple RPC invocations during path lookup and traversal. Given the performance target of 5-10 μs in NodeKernel we ultimately decided to adopt the simpler partitioning scheme where node paths are always local to a metadata server.

3.2.3 Hardware-accelerated storage

NodeKernel's data plane is designed to work well with modern networking and storage hardware. The goal is to keep the storage interface simple to avoid excessive software overheads and permit as much of the data access functionality to be implemented in hardware. Clients in NodeKernel interact with local and remote storage servers via two interfaces: `read(blockid, offset, length, buffer)` to fetch a certain number of bytes from a block, and `write(blockid, offset, buffer, length)` to write a data stored in a buffer to a block. Later in Section 4 we show that `read` and `write` operations in Crail can almost completely be offloaded to networking and storage hardware for both DRAM and Flash. Also note that the storage interface explicitly supports byte addressable storage hardware by defining the access granularity at the byte level as opposed to block level.

3.2.4 Tiered storage

NodeKernel employs a simple tiered storage design to accommodate large datasets that cannot be stored in DRAM in a cost-effective manner. Storage servers are grouped into different classes (see Figure 2), typically a class per storage technology (DRAM, NVMe SSDs, HDD, etc.). A storage server is a logical entity, i.e., one physical or virtual machine may host multiple storage servers of different types. For instance, a common deployment is to run two storage servers per host, one exporting some amount of local DRAM and one exporting storage space on the local NVMe SSD. In principle, storage classes are user-defined sets of storage servers.

Object type	Methods
Crail	**create**<T extends Node>(*path*, *sc*) → future(*T*)
	creates a data node of type T
	lookup<T extends Node>(*path*) → future(*T*)
	lookup of an existing node
	delete(*path*) → future(*boolean*)
	deletes an existing node at the given *path*
	move(*src*, *dst*) → future(*boolean*)
	moves node *src* to new location *dst*
File & KeyValue	**read**(*offset*, *buffer*, *length*) → future(*Int*)
	reads data at given offset
	append(*buffer*, *length*) → future(*Int*)
	appends application buffer to data
Bag	**read**(buffer, length) → future(*Int*)
	sequentially reads through all subfiles
Directory & Table	**enumerate**() → iterator(*Node*)
	enumerates all nodes in a given directory

Table 2: Application programming interface of Crail.

A storage server always belongs to exactly one storage class. In our evaluation we configure two storage classes, one for DRAM servers and one of NVMe SSD servers.

The traditional approach to storage tiering is to migrate data to more cost effective storage classes as the faster storage classes fill up. We found this strategy to be ineffective for temporary data due to the short lifetime of the data. Instead, we opted for a simpler approach where storage classes are filled up according to a user-defined order – typically DRAM first followed by Flash and hard disk – without ever migrating data between the tiers. Specifically during data write operations, metadata servers try to allocate DRAM blocks first, turning to lower priority storage tiers only once no higher priority storage blocks are available across the entire cluster.

4 Crail

Crail is a concrete implementation of the NodeKernel architecture. We implemented Crail in about 10K lines of Java and C++ code. Table 2 shows the application interface of Crail. The top level data type in Crail is `CrailStore`. Applications use `CrailStore` to `create`, `lookup` and `delete` data nodes, or to `move` data nodes to a different location in the storage hierarchy. Nodes are identified using path names similar to file systems. When creating a new node using `create`, applications may choose a preferred storage class for the data to be stored (parameter "sc" in Table 2). We also refer to the storage class preference as storage affinity because it allows users to specify affinity for a particular set of data to a particular set of storage servers or storage media.

Crail implements the full set of node types discussed in Section 3.1. Note that all operations in Crail are non-blocking and asynchronous (returning a `future` object). Crail's asyn-

chronous API matches well with asynchronous software interfaces for modern networking and storage hardware. In fact, almost all of Crail's high-level operations can be mapped directly to a set of non-blocking and asynchronous network and storage operations. Failures of Crail API calls are communicated either via invalid `futures` or exceptions (not shown in Table 2). For instance, an attempt to `lookup` a node that does not exist will result in an invalid `future` (`nullpointer`), while an attempt to `read` data from a node beyond the node's capacity will result in an exception.

Crail can be operated either as a shared storage service or in the form of per-user or per-application deployments. The current implementation of Crail, however, does not provide any tools to virtualize, protect and isolate multiple tenants from each other.

4.1 Metadata plane

Crail metadata servers maintain an in-memory representation of (a) the storage hierarchy, (b) the set of free-blocks, and (c) the assignments of blocks to "nodes". Specifically, each metadata server maintains a per storage class list of free blocks. Storage classes are ordered according to a user-defined preference. As discussed in Section 3.2, if during a write operation the current write position does not yet point to an allocated block, a client requests a fresh new block by calling the metadata `map` RPC operation. The metadata server selects a free block based on the selected storage affinity (or "sc" in Table 2). If there are no free blocks in the selected storage class, the metadata server attempts to allocate a block from the next storage class in the priority list. When selecting a block in a storage class, the metadata server uses round-robin over all storage servers in the given storage class to make sure data is distributed uniformly in the cluster. If no free block is available in any of the storage classes the write operation at the client will fail.

Crail partitions metadata across an array of metadata servers as discussed in Section 3.2.1, meaning, each metadata server is responsible for a partition of the storage hierarchy. Each individual server is implemented as a lightweight RPC service using DaRPC [31], an asynchronous low-latency RPC library based on RDMA send/recv. To achieve high throughput, client connections are partitioned across the different CPU cores. Each core manages a subset of the client connections in-place within a single process context to avoid context-switching overheads. All the memory for RPC buffers is allocated local to the NUMA node associated with the given CPU core that is responsible for the particular connection. Figure 3 illustrates the different aspects of the metadata processing in Crail.

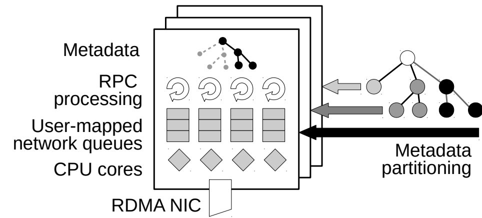


Figure 3: Lightweight metadata plane in Crail.

Enumeration: In Section 3.1 we discussed how container nodes (`Table`, `Directory`, `Bag`) maintain a list of the names of all child nodes as part of their data. The rational behind this design is that it allows us to implement container enumeration efficiently in the data plane. We have seen use cases where storing Spark shuffle data in Crail generated close to hundred thousand `File` nodes in a `Bag`. Implementing enumeration at the metadata server would lead to substantial data transfers between clients and metadata servers and in many cases would require multiple rounds of RPC to enumerate all of the nodes.

Crail's file format for container nodes is structured as an array of fixed-size records consisting of the children's name component along with a valid flag. Upon creating a new node, the metadata server assigns a unique offset within the array based on which the client writes the corresponding record. During a delete operation, after the metadata server has removed the node entry, a client clears the valid flag of the corresponding record (by zeroing the valid bit in the record). Note that the node record inside a container's data is only considered supplementary information. The metadata server always serves as the authority for validating the existence of a node. Thus, an enumerate operation running concurrently with a delete operation may lead to a situation where a node's record in the directory file is still valid, but the node's metadata state at the metadata server has already been deleted. In that case, the node is considered deleted and no read or write operations will be permitted.

4.2 Data plane

Crail implements two storage classes, one for DRAM and one for NVMe-based SSDs. An implementation of a storage class consists of a server part exporting a storage resource (see Section 3.2.3), and a client part implementing efficient data access.

RDMA storage class: A storage server in the RDMA storage class exports large regions of RDMA registered memory to the metadata server. Metadata for RDMA based storage blocks contains the necessary RDMA credentials such as address, length and stag and allows clients to directly read or write a storage block using RDMA one-sided read/writes.

NVMe-over-Fabrics storage class: NVMe-over-Fabrics (NVMf) is a recent extension to the NVMe standard that enables access to remote NVMe devices over RDMA-capable

networks. It eliminates unnecessary protocol translations along the I/O path to a remote device, exposing the multiple paired queue design of NVMe directly to clients. As with RDMA, the queue pairs can directly be mapped into the application context to avoid kernel overheads.

A Crail NVMf storage server acts as a control plane for a NVMf controller by connecting to the controller and reporting its credentials like NVMe qualified name, size, etc., to the metadata server. With the credentials provided by the metadata server, the clients can directly connect to the NVMf controller and perform block read and write operations.

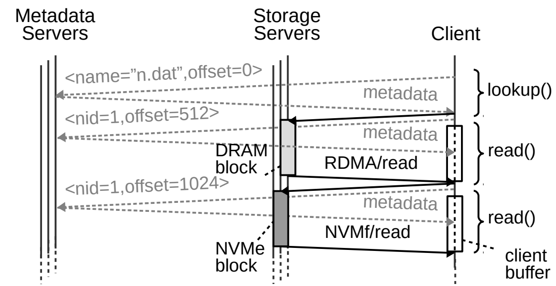


Figure 4: Anatomy of file read/write operations in Crail.

4.3 Failure semantics and persistence

Crail does not currently implement mechanisms for fault-tolerance (see Section 3.2) and therefore does not protect against machine or hardware failures. On a crash of a storage server the corresponding data blocks are lost. On a crash of a metadata server the correponding metadata partition is lost. Metadata servers remove inaccessible storage servers from the list of active servers based on keep-alive messages and make sure only active servers are considered during block allocation.

Crail provides optional mechanisms to persist data stored in DRAM, shut down a Crail deployment and to start a Crail deployment from a previously persisted state. Persistence is implemented via operation logging at the metadata servers and the use of memory mapped persistent storage at storage servers.

4.4 Anatomy of data access

Figure 4 illustrates how a Crail client interacts with storage and metadata servers on behalf of an application reading data from a `File` or `KeyValue` node. The application first calls `lookup` to retrieve a node handle, causing Crail to fetch the necessary metadata via RPC from the metadata server. The metadata contains information about the node such as the size of the data and the location of the first block. Following a successful `lookup` call, the application issues a `read` operation to read a certain number of bytes from the node. The requested number of bytes may be less than a block. In that case a single RDMA or NVMf operation will be sufficient to complete the request. If the requested number of bytes spawns multiple blocks, as it is case in the example, Crail immediately issues the data transfer for the first block, while in parallel requesting the metadata for the next block. Under normal circumstances – due to the low latency RDMA-based protocol between the client and the metadata server – the metadata request will complete ahead of the current block transfer and guarantee a continued data transfer without the client ever having to wait for missing metadata information.

5 Evaluation

In our evaluation we assess if Crail meets the requirements for a temporary storage platform discussed in Section 2.1. Specifically we answer the following questions:

1. Does the unified abstraction of Crail, with its extra indirection layer, perform well for a wide spectrum of data sizes on high-performance devices? (Section 5.1)

2. How simple is it to map higher-level workloads (with their temporary data accesses) to Crail? (Section 5.2)

3. How big are the performance and cost benefits of a mixed-media storage system for data-processing frameworks? (Section 5.3)

Cluster configuration: We use a cluster of eight x64 nodes with two Intel(R) Xeon(R) CPU E5-2690 v1 @ 2.90GHz CPUs, 96GB DDR3 DRAM and a 100 Gbit/s Mellanox ConnectX-5 RoCE RDMA network card. For the client server microbenchmarks, the server is configured with 4 Intel Optane 900P SSDs, except for the IOPS experiments where we only use 2 Optane drives per server. For the larger cluster experiments, all 8 nodes are equipped with 4 Samsung 960 Pro SSDs. The nodes run Ubuntu 16.04.3 LTS (Xenial Xerus) with Linux kernel version 4.10.0-33-generic and Java 8.

5.1 Microbenmarks

Small and medium-size values: We first start by evaluating Crail's performance for storing small to medium size values, a use case typically well served by key-value stores. Consequently, we compare Crail's performance (latency and IOPS) with two state-of-the-art open-source key-value stores, namely, RAMCloud (for DRAM storage) and Aerospike (for NVM Optane). Figure 5 shows the performance for `get` and `put` operations for different data sizes. In Crail, a put operation is implemented by creating a `KeyValue` node using the `create` API call (see Table 2), followed by an `append` operation. The get operation is implemented using a `lookup` call followed by a `read` on the `KeyValue` node – similar to

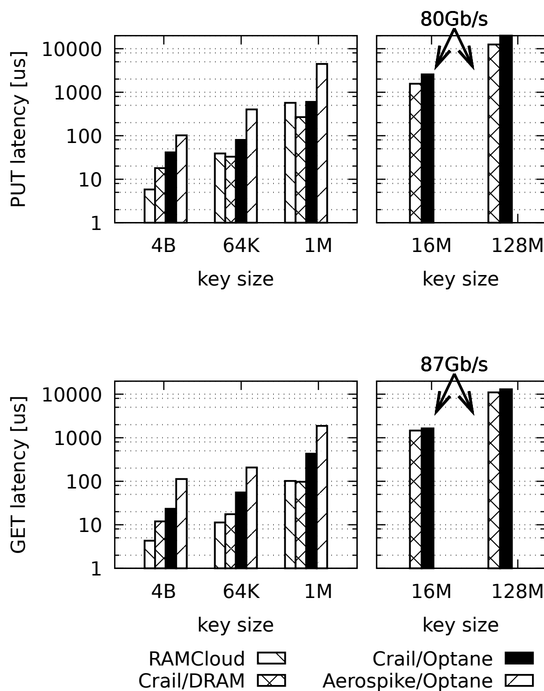


Figure 5: Put/Get latencies in Crail, RAMCloud and Aerospike for datasets of different sizes.

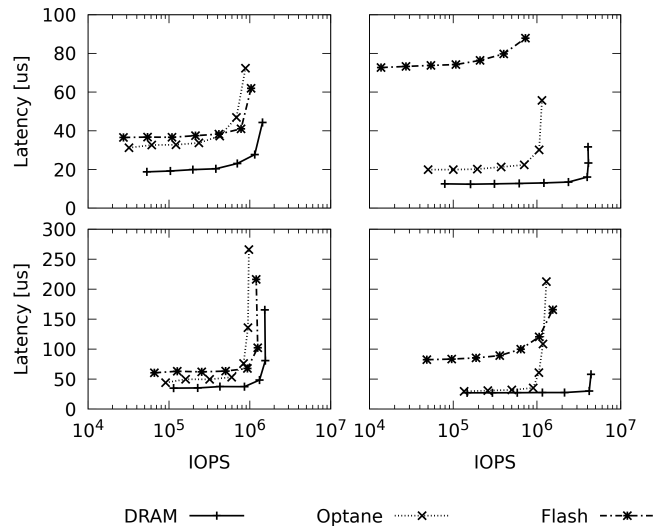


Figure 6: Media-specific loaded latency profile. Top left: Put, queue depth 1. Top right: Get, queue depth 1. Bottom left: Put, queue depth 4. Bottom right: Get, queue depth 4.

the scenario shown in Figure 4. For small datasets (4 bytes), Crail performs slightly worse than RAMCloud for DRAM storage (12 μs vs 6 μs), but outperforms Aerospike on Optane NVM by a margin of $2-4\times$ (23-40 μs for Crail vs. 100 μs for Aerospike). The difference between Aerospike and Crail comes from differences in their I/O execution. Aerospike uses synchronous I/O and multiple I/O threads, which cause contention and spend a significant amount of execution time in synchronization functions [18]. Crail uses asynchronous I/O and executes I/O requests in one context, avoiding context switching and synchronization completely. The latency difference between Crail and RAMCloud is acceptable considering that RAMCloud is a system optimized for small values. For medium sized values (64KB-1MB), Crail outperforms RAMCloud and Aerospike by a margin of $2-6.8\times$. For instance, a 1MB Put on Crail takes around 590 μs, versus 4 *ms* in Aerospike. These performance gains come from the efficient use of RDMA one-sided operations (for both DRAM and NVM), which eliminates data copies at both client and server ends and generally reduces the code path that is executed during put/get operations. While Crail natively supports arbitrary size datasets – by distributing the blocks over multiple storage servers – storing such large values in systems like RAMCloud or Aerospike is either difficult or prohibited. For instance, Aerospike limits individual key/value pairs to 1MB. RAMCloud does not have a strict size limitation but failed to store values larger than 4MB.

For sake of completeness, Figure 5 also shows put/get latencies for extra large values of 16MB and 128MB. As we can see, it takes around to 12 *ms* to store a 128MB value in Crail's DRAM tier, and 20 *ms* to store the same dataset in Crail's Optane tier. Storing such large values in Crail is entirely a matter of throughput. Consequently, the remote DRAM latency is limited by the 100 Gb/s network bandwidth, while the NVM latency is the determined by the bandwidth of the storage device. The aggregated bandwidth of the 4 Optane drives is around 10-12 GB/s. Hence, the resulting data access bandwidth for large datasets stored in Crail's NVM tier is 80-87 Gb/s.

IOPS scaling: So far we have discussed unloaded latencies. Figure 6 shows the latency profile for 256 bytes values for a loaded Crail system for different media types. In this setup, we increase the number of clients from 1 to 64. The clients are running on 16 physical machines, issuing put/get operations in a tight loop. We use only one storage server and one metadata server in this setup, configured either to serve DRAM, Optane NVM or Flash. The top row in Figure 6 show the case for a queue depth of 1, meaning, each client always has only one operation in flight. As shown in the figure, Crail delivers stable latencies up to a reasonably high throughput. For DRAM, the get latencies (top right in Figure 6) stay at 12-15 μs up to 4M IOPS, at which point the metadata server became the bottleneck. We ran the same experiment with multiple metadata servers and verified that the system throughput was scaling linearly (shown later in Figure 7 on top). For the Optane NVM configuration, latencies stay at 20 μs up until almost 1M IOPS, which is very close to the device limit. The Flash latencies are higher but the Samsung drives also have a higher throughput limit. In fact, 64 clients with queue depth 1 cannot saturate the Samsung devices. In order to generate

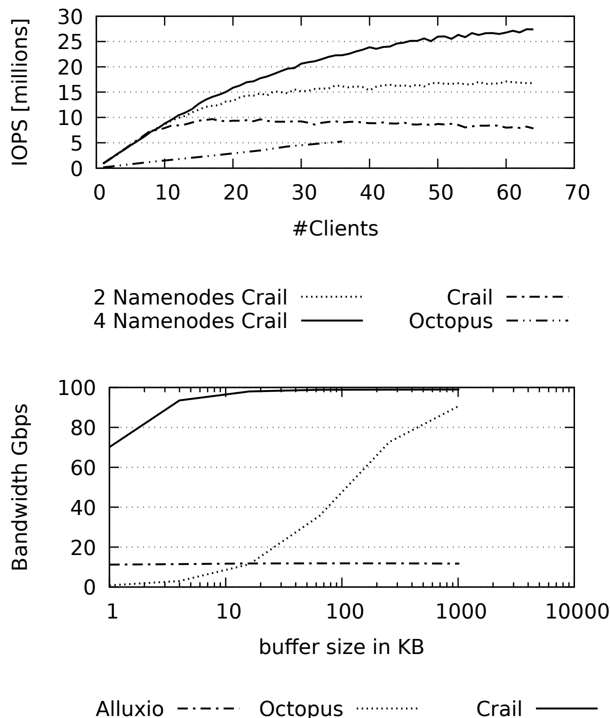


Figure 7: Top: metadata performance in Crail compared to Octopus (we could not run Octopus reliably after 36 clients). Bottom: sequential read performance in Crail and Octopus for large datasets and different buffer sizes.

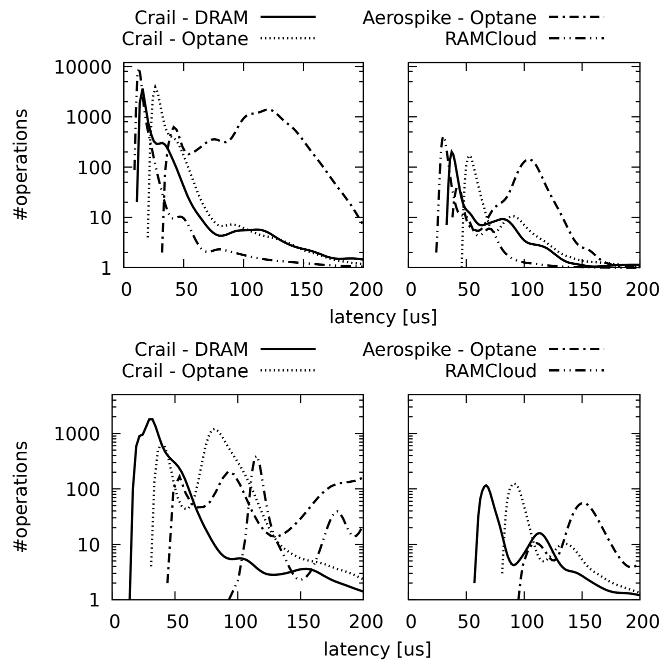


Figure 8: YCSB benchmark performance. Top: small value (1K per KV pair) read (left) and update (right) latencies. Bottom: large value (100K per KV pair) read and update latencies.

a higher load, we measured throughput and latencies for the case where each client always has four operations in flight (queue depth 4, bottom row in Figure 6). As shown, queue depth 4 generally achieves a higher throughput up to a point where the hardware limit is reached, the device queues are overloaded (e.g., for NVM Optane) and latencies sky rock. For instance, at the point before the exponential increase in the latencies, Crail delivers get latencies (Figure 6 bottom right) of 30.1 μs at 4.2M IOPS (DRAM), 60.7 μs for 1.1M IOPS (Optane), and 99.86 μs for 640.3K IOPS (Flash). The situation for put is similar, though generally with lower performance.

Metadata performance: In Figure 7 (top), we benchmark the performance of a simple `lookup` metadata operation which is used to retrieve node metadata in Crail, and compare it with the performance of a similar metadata operation `getattr` in Octopus [23] (an RDMA-optimized NVM file system running in DRAM). There are two main observations here. First, for the single namenode case, Crail outperforms Octopus by $1.7-5.9\times$. A single namenode in Crail peaks around 9.3M lookups/sec. Second, Crail can very efficiently scale the single namenode performance to multi-namenode setups. The system can deliver up to 16.7M and 27.4M lookups/sec for 2 and 4 namenode configurations.

Accessing large datasets: Figure 7 (bottom) shows the bandwidth (y-axis) measured when reading large datasets of `File` nodes in Crail, in comparison to file read operations in Octopus and Alluxio. The x-axis in Figure 7 refers to the size of the application buffer the client is using during read operations. Crail with its efficient data and metadata plane, and overlapping of lookup RPCs and data fetching quickly reaches the network bandwidth limit even for relatively small buffer sizes (just over 1kB). Alluxio performance is bottlenecked by the CPU due to data copies and inefficiencies in the network stack implementation. Octopus performs better than Alluxio, and gradually for large buffers (close to 1MB) reaches the line speed of 98 Gb/s. Note that Crail's peak bandwidth (98 Gb/s) in Figure 7 is better than the peak bandwidth (87 Gb/s) in Figure 5 because for the KV experiments each `KeyValue` node is opened, read, and closed whereas for the file experiments those accesses are amortized.

Summary: In this section we have shown that Crail can store a large spectrum of values effectively while offering comparable or superior performance than the other state-of-the-art systems that are optimized for a particular data range.

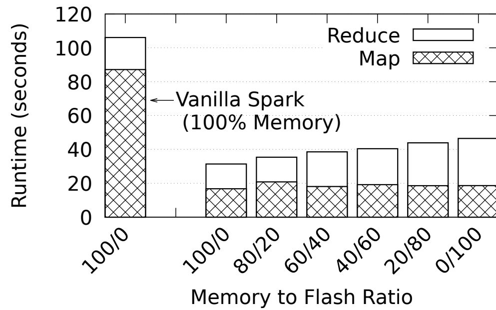


Figure 9: Spark Terasort with mixed DRAM-NVM.

5.2 Systems-level Benchmarks

5.2.1 NoSQL workloads

The Yahoo! Cloud Serving Benchmark (YCSB) is an open standard designed to compare the performance of NoSQL databases [10]. It comes with five workloads that stress different properties, e.g. workload A is update heavy whereas workload C is read heavy. We choose workload B to compare Crail to RAMCloud and Aerospike. Workload B has 95% read and 5% update operations and the records are selected with a Zipfian distribution. All systems run in a single namenode/datanode configuration. The purpose of this experiment is to evaluate the latency profile of Crail for a realistic workload beyond microbenchmarks presented in the last section.

Figure 8 (top part) shows the read and update latency distributions for workload B using a single client for the default setup of 10 fields of 100 bytes per record (1K per KV pair). The left part of the figure shows the read performance, and the right part shows the update performance. As shown for this default setup, the largest number of read operations observe a latency of 14 μs (95% and 99% percentile are at 37 and 84 μs) and 26 μs (95% and 99% percentile are at 47 and 81 μs) for DRAM and Optane respectively. Crail on Optane has an average latency of 38 μs and thus is only 15 μs slower than Crail on DRAM. On the other hand, Aerospike with Optane delivers an average latency of 108.7 μs, which is 2.84$\times$ worse than the average Crail latency (38.03 μs). Comparing Crail's DRAM performance to RAMCloud shows that RAMCloud is slightly faster than Crail. However, as we move to larger values of 10 fields of 10KB each (100KB per KV pair) in Figure 8 (bottom) Crail is almost $2.6-4.8\times$ better than Aerospike and RAMCloud, respectively.

Summary: Our experiments using the YCSB benchmark have demonstrated that (a) Crail can successfully translate the raw DRAM/NVMe performance advantages into workload level gains, and (b) Crail effectively deals with both small and large datasets while RAMCloud and Aerospike perform their best in a specific operating range.

5.2.2 Spark Integration

We present the evaluation of Crail with Spark, one of the most popular data processing engines. Spark executes workloads as a series of map-reduce steps while sharing performance critical data in each step. There are multiple points in the Spark data processing pipeline where temporary datasets are generated. In this section, we show performance measurements specifically for two: shuffle and broadcast. Both subsystems can easily be implemented as plugin modules for Spark.

Broadcast: We implemented broadcast using Crail by storing broadcast data as `KeyValue` nodes in a non-enumerable `Table`. A broadcast writer creates a new `KeyValue` node, appends the broadcast data to the node, and passes the node "name" to the readers. Readers, which are distributed over multiple machines inside Spark executors, do a lookup on the "name" and read the data from Crail. Figure 10a shows the result. The x-axis shows the latency as observed by different broadcast readers in the Spark job, while the y-axis shows the percentage of readers. The solid vertical line represent the baseline latency of 12 μs, which we demonstrated in our microbenchmarks. As shown, most of the Crail broadcast readers observe latency very close to the minimum possible. A few observe a latency lower than 12 μs because some of these readers are co-located on the same physical machine where the values are stored. For those nodes, even though they still read the data using the local network interface, there is no actual network transfer happening, hence, their read performance is not limited by the network. In summary, Crail broadcast performance is 1-2 orders of magnitude better than the default Spark implementation.

Shuffle: In Spark, a shuffle writer continuously generates shuffle data during the map phase as it processes the input dataset and classifies data into different buckets that are later read by reducers. Due to the large fan-in and fan-out access pattern, we implemented shuffle using Crail `Bag` nodes. There is one `Bag` node per reducer and each shuffle writer appends data to an array of privately owned `File` nodes, one `File` node per writer per bag. After the map phase, each reducer reads its associated bag using the optimized read interface available in the `Bag` node type (see Section 3.1). We generated a large amount of data (512 GB) and triggered the shuffle operation using the GroupBy benchmark available in the Spark source code. Figures 10b and 10c show the performance (runtime on the x-axis) and the observed network throughput (y-axis) for various configurations. The values 1, 4, or 8 represent the number of cores given to each Spark executor. A quick comparison of the two figures shows that the Crail-accelerated Spark observes higher network throughput (for a corresponding core count) and, thus, as a result better runtimes (1 core 5x, 4 cores 2.5x and 8 cores 2x).

Summary: In this section we have demonstrated that Crail is able to successfully accelerate temporary data access in Spark for small values (e.g., broadcast) as well as large values

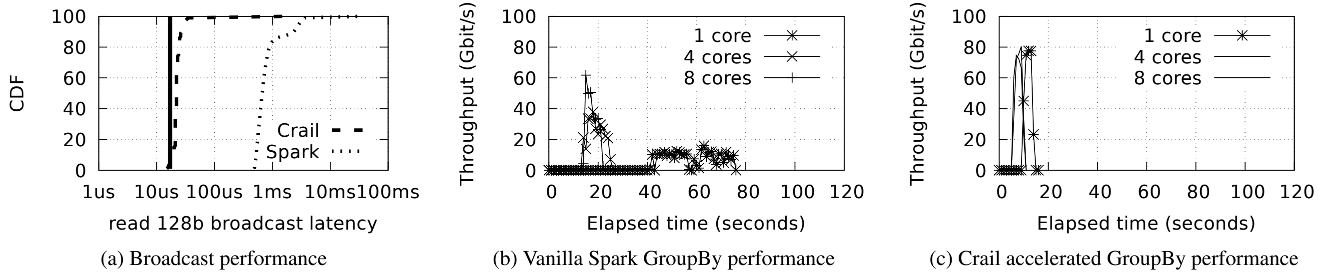


Figure 10: Spark broadcast and GroupBy performance, using Vanilla Spark vs. Crail temporary storage.

(e.g., shuffle) by taking advantage of the different node types (`KeyValue` and `Bag`) in Crail.

5.3 Efficiency of hybrid DRAM/NVM setup

As the last part of our evaluation, we quantify how Crail's tiered data plane helps with regard to performance and cost objectives. We consider the Terasort workload, which is one of the most I/O intensive applications on Spark. We implement Terasort as an external range-partition sort algorithm in two stages. The first stage maps the incoming key-value pairs (10 bytes keys and 90 bytes value) into external buckets. These buckets are then shuffled and sorted by individual reduce tasks. For this evaluation we use the accelerated shuffle and broadcast plugins that we previously developed.

In Figure 9, we explore the performance/cost trade-off of using NVM instead of DRAM to store shuffle data in a 200 GB Spark sorting workload. For this we configure Crail with different storage limits for the DRAM and the Flash storage tiers. The x-axis indicates what fraction of the total shuffle data is stored in DRAM versus Flash. Note that in this experiment we are using the Samsung Flash-based SSDs rather than the Optane devices. A configuration of 10/90 means that 10% of the data is held in DRAM, while 90% is held in Flash. The figure also shows the performance of vanilla Spark (first bar of the figure) that runs on its default shuffle engine completely in DRAM (using tmpfs as a storage backend). There are two key observations here. First, in comparison to vanilla Spark, the use of Crail for the shuffle backend already reduces the runtime by a factor of 3.4. This performance gain can be attributed to the efficient use of high-performance networking and storage hardware in Crail. For instance, during the reduce phase we measured an all-to-all network throughput of 70 Gb/s/machine. Second, as we decrease the fraction of DRAM in Crail in favor of Flash, Spark graciously and automatically spills shuffle data into the Flash tier. In the extreme configuration, where all shuffle data is stored in Flash, the performance degrades to 46.49 second (48% increase), while to total cost for storage is reduced by 8× from 1,000$ to 126$ (to store 200 GB of data based on the numbers in Table 1). The gradual spilling of data from DRAM to Flash happens transparently. Even in the all-Flash configuration, the performance of the Crail-integrated Spark Terasort is half of the completely-in-DRAM vanilla Spark performance. These results validate the design choices we made in Crail that permit trading performance for storage cost.

Summary: In this section, we demonstrated that the use of Crail in Spark (i) leads to better performance due to its efficient I/O path; (ii) reduces the cost of storage, and increases the performance due the hybrid DRAM-NVMe architecture.

6 Conclusion

Storing and accessing temporary data efficiently in data processing workloads is critical for performance, yet challenging due to complex storage demands that fall between the lines of existing storage systems like file systems or key-value stores. We presented NodeKernel, a novel storage architecture offering a new point in the storage design space by combining hierarchical naming with scalability and excellent performance for a wide range of data sizes and access patterns that are typical for temporary data. The NodeKernel architecture is driven by opportunities of modern networking and storage hardware that enabled us to reduce overheads that made such a design impractical in the past. We showed that storing temporary data in Crail, our concrete implementation of the NodeKernel architecture leveraging RDMA networking and NVMe storage, can improve NoSQL workloads by up to 4.8× and Spark application performance by up to 3.4×. Crail's use of NVMe Flash further reduces storage cost by up to 8× compared to storage systems that only use DRAM.

Acknowledgments

We thank our shepherd, Michael Swift, and the anonymous Usenix ATC reviewers for their helpful feedback.

References

[1] Alluxio: Open source memory speed virtual distributed storage. https://www.alluxio.org/.

[2] Apache Crail (Incubating). http://crail.apache.org/.

[3] Apache Crail (Incubating) Source Code. https://github.com/apache/incubator-crail.

[4] Memcached. http://memcached.org.

[5] Redis. https://redis.io/.

[6] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, 2018.

[7] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, January 2017.

[8] Matt Benjamin. Xiomessenger: Ceph transport abstraction based on accelio, a high-performance message-passing framework by mellanox, at https://www.cohortfs.com/ceph-over-accelio.

[9] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 20:1–20:15, New York, NY, USA, 2018. ACM.

[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 '10, pages 143–154, New York, NY, USA, 2010.

[11] Aaron Davidson and Andrew Or. Optimizing shuffle performance in spark. In *https://pdfs.semanticscholar.org/d746/505bad055c357fa50d394d15eb380a3f1ad3.pdf*, 2013.

[12] 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, 2014.

[13] 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)*, Boston, MA, 2019.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, 2014.

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016.

[16] 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, GA, 2016.

[17] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, Berkeley, CA, USA, 2018.

[18] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, 2019. USENIX Association.

[19] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017.

[20] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011.

[21] 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 Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014.

[22] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 48–63, New York, NY, USA, 2017.

[23] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 773–785, Berkeley, CA, USA, 2017.

[24] Chenxin Ma, Virginia Smith, Martin Jaggi, Michael I. Jordan, Peter Richtárik, and Martin Takáč. Adding vs. averaging in distributed primal-dual optimization. In *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ICML'15, pages 1973–1982, 2015.

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, 2018.

[26] 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 ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[27] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012.

[28] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012.

[29] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press.

[30] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[31] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 15:1–15:13, New York, NY, USA, 2014.

[32] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 615–630, Boston, MA, 2018.

[33] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017.

[34] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.*, 10(4):301–312, November 2016.

[35] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. In *IEEE Bulletin of the Technical Committee on Data Engineering*, pages 39–45, 2018.

[36] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012.

[37] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other products and service names might be trademarks of IBM or other companies.

Evaluating File System Reliability on Solid State Drives

Shehbaz Jaffer* University of Toronto
Stathis Maneas* University of Toronto
Andy Hwang University of Toronto
Bianca Schroeder University of Toronto

Abstract

As solid state drives (SSDs) are increasingly replacing hard disk drives, the reliability of storage systems depends on the failure modes of SSDs and the ability of the file system layered on top to handle these failure modes. While the classical paper on IRON File Systems provides a thorough study of the failure policies of three file systems common at the time, we argue that 13 years later it is time to revisit file system reliability with SSDs and their reliability characteristics in mind, based on modern file systems that incorporate journaling, copy-on-write and log-structured approaches, and are optimized for flash. This paper presents a detailed study, spanning ext4, Btrfs and F2FS, and covering a number of different SSD error modes. We develop our own fault injection framework and explore over a thousand error cases. Our results indicate that 16% of these cases result in a file system that cannot be mounted or even repaired by its system checker. We also identify the key file system metadata structures that can cause such failures and finally, we recommend some design guidelines for file systems that are deployed on top of SSDs.

1 Introduction

Solid state drives (SSDs) are increasingly replacing hard disk drives as a form of secondary storage medium. With their growing adoption, storage reliability now depends on the reliability of these new devices as well as the ability of the file system above them to handle errors these devices might generate (including for example device errors when reading or writing a block, or silently corrupted data). While the classical paper by Prabhakaran et *al.* [45] (published in 2005) studied in great detail the robustness of three file systems that were common at the time in the face of hard disk drive (HDD) errors, we argue that there are multiple reasons why it is time to revisit this work.

The first reason is that failure characteristics of SSDs differ significantly from those of HDDs. For example, recent field studies [39, 43, 48] show that, while their replacement rates (due to suspected hardware problems) are often by an order of magnitude lower than those of HDDs, the occurrence of partial drive failures that lead to errors when reading or writing a block or corrupted data can be an order of magnitude higher. Other work argues that the Flash Translation Layer (FTL) of SSDs might be more prone to bugs compared to HDD firmware, due to their high complexity and less maturity, and demonstrate this to be the case when drives are faced with power faults [53]. This makes it even more important than before that file systems can detect and deal with device faults effectively.

Second, file systems have evolved significantly since [45] was published 13 years ago; the ext family of file systems has undergone major changes from the ext3 version considered in [45] to the current ext4 [38]. New players with advanced file-system features have arrived. Most notably Btrfs [46], a copy-on-write file system which is more suitable for SSDs with no in-place writes, has garnered wide adoption. The design of Btrfs is particularly interesting as it has fewer total writes than ext4's journaling mechanism. Further, there are new file systems that have been designed specifically for flash, such as F2FS [33], which follow a log-structured approach to optimize performance on flash.

The goal of this paper is to characterize the resilience of modern file systems running on flash-based SSDs in the face of SSD faults, along with the effectiveness of their recovery mechanisms when taking SSD failure characteristics into account. We focus on three different file systems: Btrfs, ext4, and F2FS. ext4 is an obvious choice, as it is the most commonly used Linux file system. Btrfs and F2FS include features particularly attractive with respect to flash, with F2FS being tailored for flash. Moreover, these three file systems cover three different points in the design spectrum, ranging from journaling to copy-on-write to log-structured approaches.

The main contribution of this paper is a detailed study, spanning three very different file systems and their ability to detect and recover from SSD faults, based on error injection targeting all key data structures. We observe huge differences across file systems and describe the vulnerabilities of each in detail.

*These authors contributed equally to this work.

Over the course of this work we experiment with more than one thousand fault scenarios and observe that around 16% of them result in severe failure cases (kernel panic, unmountable file system). We make a number of observations and file several bug reports, some of which have already resulted in patches. For our experiments, we developed an error injection module on top of the Linux device mapper framework.

The remainder of this paper is organized as follows: Section 2 provides a taxonomy of SSD faults and a description of the experimental setup we use to emulate these faults and test the reaction of the three file systems. Section 3 presents the results from our fault emulation experiments. Section 4 covers related work and finally, in Section 5, we summarize our observations and insights.

2 File System Error Injection

Our goal is to emulate different types of SSD failures and check the ability of different file systems to detect and recover from them, based on which part of the file system was affected. We limit our analysis to a local file system running on top of a single drive. Note that although multi-drive redundancy mechanisms like RAID exist, they are not general substitutes for file system reliability mechanisms. First, RAID is not applicable to all scenarios, such as single drives on personal computers. Second, errors or data corruption can originate from higher levels in the storage stack, which RAID can neither detect nor recover.

Furthermore, our work only considers *partial* drive failures, where only part of a drive's operation is affected, rather than *fail-stop* failures, where the drive as a whole becomes permanently inaccessible. The reason lies in the numerous studies published over the last few years, using either lab experiments or field data, which have identified many different SSD internal error mechanisms that can result in partial failures, including mechanisms that originate both from the flash level [10, 12, 13, 16–19, 21, 23, 26, 27, 29–31, 34, 35, 40, 41, 47, 49, 50] and from bugs in the FTL code, e.g. when it is not hardened to handle power faults correctly [52, 53].

Moreover, a field study based on Google's data centers observes that partial failures are significantly more common for SSDs than for HDDs [48].

This section describes different SSD error modes and how they manifest at the file system level, and also our experimental setup, including the error injection framework and how we target different parts of a file system.

2.1 SSD Errors in the Field and their Manifestation

This section provides an overview over the various mechanisms that can lead to partial failures and how they manifest at the file system level (all summarized in Table 1).

Uncorrectable Bit Corruption: Previous work [10, 12, 13, 16–19, 21, 26, 27, 29, 34, 35, 41] describes a large number of error mechanisms that originate at the flash level and can result in *bit corruption*, including retention errors, read and program disturb errors, errors due to flash cell wear-out and failing blocks. Virtually all modern SSDs incorporate error correcting codes to detect and correct such bit corruption. However, recent field studies indicate that *uncorrectable bit corruption*, where more bits are corrupted than the error correcting code (ECC) can handle, occurs at a significant rate in the field. For example, a study based on Google field data observes 2-6 out of 1000 drive days with uncorrectable bit errors [48]. Uncorrectable bit corruption manifests as a read I/O error returned by the drive when an application tries to access the affected data ("Read I/O errors" in Table 1).

Silent Bit Corruption: This is a more insidious form of bit corruption, where the drive itself is not aware of the corruption and returns corrupted data to the application ("Corruption" in Table 1). While there have been field studies on the prevalence of silent data corruption for HDD based systems [9], there is to date no field data on silent bit corruption for SSD based systems. However, work based on lab experiments shows that 3 out of 15 drive models under test experience silent data corruption in the case of power faults [53]. Note that there are other mechanisms that can lead to silent data corruption, including mechanisms that originate at higher levels in the storage stack, above the SSD device level.

FTL Metadata Corruption: A special case arises when silent bit corruption affects FTL metadata. Among other things, the FTL maintains a mapping of logical to physical (L2P) blocks as part of its metadata [8]; metadata corruption could lead to "Read I/O errors" or "Write I/O errors", when the application attempts to read or write a page that does not have an entry in the L2P mapping due to corruption. Corruption of the L2P mapping could also result in wrong or erased data being returned on a read, manifesting as "Corruption" to the file system. Note that this is also a silent corruption - i.e. neither the device nor the FTL is aware of these corruptions.

Misdirected Writes: This refers to the situation where during an SSD-internal write operation, the correct data is being written to flash, but at the wrong location. This might be due to a bug in the FTL code or triggered by a power fault, as explained in [53]. At the file system level this might manifest as a "Corruption", where a subsequent read returns wrong data, or a "Read I/O error". This form of corruption is silent; the device does not detect and propagate errors to the storage stack above until invalid data or metadata is accessed again.

Shorn Writes: A shorn write is a write that is issued by the file system, but only partially done by the device. In [53], the authors observe such scenarios surprisingly frequently during power faults, even for enterprise class drives, while issuing properly synchronized I/O and cache flush commands to the device. A shorn write is similar to a "torn write", where only part of a multi-sector update is written to the disk, but it applies to sector(s) which should have been fully persisted due to the use of a cache flush operation. One possible expla-

nation is the mismatch of write granularities between layers. The default block size for file systems is larger (e.g. 4KB for ext4/F2FS, and 16KB for Btrfs) than the physical device (e.g. 512B). A block issued from the file system is mapped to multiple physical blocks inside the device. As a result, during a power fault, only some of the mappings are updated while others remain unchanged. Even if physical block sizes match that of the file system, another possible explanation is because SSDs include on-board cache memory for buffering writes, shorn writes may also be caused by alignment and timing bugs in the drive's cache management [53]. Moreover, recent SSD architectures use pre-buffering and striping across independent parallel units, which do not guarantee atomicity between them for an atomic write operation [11]. The increase in parallelism may further expose more shorn writes.

At the file system level, a shorn write is not detected until its manifestation during a later read operation, where the file system sees a 4KB block, part of which contains data from the most recent update to the block, while the remaining part contains either old or zeroed out data (if the block was recently erased). While this could be viewed as a special form of silent bit corruption, we consider this as a separate category in terms of how it manifests at the file system level (called "Shorn Write" corresponding to column (d) in Table 1) as this form of corruption creates a particular pattern (each sequence of 512 bytes within a 4KB block is either completely corrupted or completely correct), compared to the more random corruption event referred to by column (c).

In [53], the authors observe shorn writes manifesting in two patterns, where only the first 3/8th or the first 7/8th of a block gets written and the rest is not. Similarly in our experiments, we keep only the first 3/8th of a 4KB block. We assume the block has been successfully erased, so the rest of the block remains zeroed out. Our module can be configured to test other shorn write sizes and patterns as well.

Dropped writes: The authors in [53] observe cases where an SSD internal write operation gets dropped even after an explicit cache flush (e.g. in the case of a power fault when the update was in the SSD's cache, but not persisted to flash). If the dropped write relates to FTL metadata, in particular to the L2P mapping, this could manifest as a "Read I/O error", "Write I/O error" or "Corruption" on a subsequent read or write of the data. If the dropped write relates to a file system write, the result is the same as if the file system had never issued the corresponding write. We create a separate category for this manifestation which we refer to as "Lost Write" (column (e) in Table 1).

Incomplete Program operation: This refers to the situation where a flash program operation does not fully complete (without the FTL noticing), so only part of a flash page gets written. Such scenarios were observed, for example, under power faults [53]. At the file system level, this manifests as a "Corruption" during a subsequent read of the data.

Incomplete Erase operation: This refers to the situation where a flash erase operation does not completely erase a flash erase block (without the FTL detecting and correcting this problem). Incomplete erase operations have been observed under power faults [53]. They could also occur when flash erase blocks wear-out and the FTL does not handle a failed erase operation properly. Subsequent program operations to the affected erase block can result in incorrectly written data and consequently "Corruption", when this data is later read by the file system.

SSD/Flash Errors	(a)	(b)	(c)	(d)	(e)
Uncorrectable Bit Corruption	✓				
Silent Bit Corruption			✓		
FTL Metadata Corruption	✓	✓	✓		
Misdirected Writes	✓		✓		
Shorn Writes				✓	
Dropped Write	✓	✓	✓		✓
Incomplete Program Operation			✓	✓	
Incomplete Erase Operation			✓		

Table 1: *Different types of flash errors and their manifestation in the file system. (a) Read I/O error (b) Write I/O error (c) Corruption (d) Shorn Write (e) Lost Write.*

2.2 Comparison with HDD faults

We note that there are also HDD-specific faults that would manifest in a similar way at the file system level. However, the mechanisms that cause faults within each media are different and can for example affect the frequency of observed errors. One such case are uncorrectable read errors which have been observed at a much higher frequency in production systems using SSDs than HDDs [48] (a trend that will likely only get worse with QLC). There are faults though whose manifestation does actually differ from HDDs to SSDs, due to inherent differences in their overall design and operation. For instance, a part affected by a shorn write may contain previously written data in the case of an HDD block, but would contain zeroed out data if that area within the SSD has been correctly erased. In addition, the large degree of parallelism inside SSDs makes correctness under power faults significantly more challenging than for HDDs (for example, ensuring atomic writes across parallel units). Finally, file systems might modify their behavior and apply different fault recovery mechanisms for SSDs and HDDs; for example, Btrfs turns off metadata duplication by default when deployed on top of an SSD.

2.3 Device Mapper Tool for Error Emulation

The key observation from the previous section is that all SSD faults we consider manifest in one of five ways, corresponding to the five columns (a) to (e) in Table 1. This section describes a device mapper tool we created to emulate all five scenarios.

In order to to emulate SSD error modes and observe each individual file system's response, we need to intercept the block I/O requests between the file system and the block device. We leverage the Linux *device mapper* framework to create a virtual block device that intercepts requests between the file system and the underlying physical device. This allows

Programs
mount, umount, open, creat, access, stat, lstat, chmod, chown, utime, rename, read, write, truncate, readlink, symlink, unlink, chdir, rmdir, mkdir, getdirentries, chroot

Table 2: *The programs used in our study. Each one stresses a single system call and is invoked several times under different file system images to increase coverage.*

ext4	
Data Structure	**Approach**
super block, group descriptor, inode blocks, block bmap, inode bmap	dumpe2fs
dir_entry	debugfs, get block inode, stat on inode number, check file type
extent	debugfs, check for extent of a file or directory path
data	debugfs, get block inode, stat on inode number, check file type
journal	debugfs, check if parent inode number is 8
Btrfs	
Data Structure	**Approach**
fstree, roottree, csumtree, extentTree, chunkTree, uuidTree, devTree, logTree	device mapper module check btrfs node header fields at runtime
DIR_ITEM DIR_INDEX INODE_REF INODE_DATA EXTENT_DATA	btrfs-debug-tree
F2FS	
superblock, checkpoint, SIT, NAT, inode, d/ind node, dir. block, data	device mapper module

Table 3: *The approach to type blocks collected using either* `blktrace` *or our own device mapper module.*

us to operate on block I/O requests and simulate faults as if they originate from a physical device, and also observe the file system's reaction without modifying its source code. In this way, we can perform tracing, parse file system metadata, and alter block contents online, for both read and write requests, while the file system is mounted. For this study, we use the Linux kernel version 4.17.

Our module can intercept read and write requests for selected blocks as they pass through the block layer and return an error code to the file system, emulating categories (a) "Read Error" and (b) "Write Error" in Table 1. Possible parameters include the request's type (read/write), block number, and data structure type. In the case of multiple accesses to the same block, one particular access can be targeted. We also support corruption of specific data structures, fields and bytes within blocks, allowing us to emulate category (c) "Corruption". The module can selectively shear multiple sectors of a block before sending it to the file system or writing it on disk, emulating category (d) "Shorn Write". Our module can further drop one or more blocks while writing the blocks corresponding to a file system operation, emulating the last category (e) "Lost Write". The module's API is generic across file systems and can be expanded to different file systems. Our module can be found at [6].

2.4 Test Programs

We perform injection experiments while executing test programs chosen to exercise different parts of the POSIX API, similar to the "singlets" used by Prabhakaran et *al.* [45]. Each individual program focuses on one system call, such as `mkdir` or `write`. Table 2 lists all the test programs that we used in our study. For each test program, we populate the disk with different files and directory structures to increase code coverage. For example, we generate small files that are stored inline within an inode, as well as large files that use indirect blocks. All our programs pedantically follow POSIX semantics; they call `fsync(2)` and `close(2)`, and check the return values to ensure that data and metadata has successfully persisted to the underlying storage device.

2.5 Targeted Error Injection

Our goal is to understand the effect of block I/O errors and corruption in detail depending on which part of a file system is affected. That means our error injection testbed requires the ability to target specific data structures and specific fields within a data structure for error injection, rather than randomly injecting errors. We therefore need to identify for each program which data structures are involved and how the parts of the data structure map to the sequence of block accesses generated by the program.

Understanding the relationship between the sequence of block accesses and the data structures within each file system required a significant amount of work and we had to rely on a combination of approaches. First, we initialize the file system to a clean state with representative data. We then run a specific test program (Table 2) on the file system image, capturing traces from `blktrace` and the kernel to learn the program's actual accessed blocks. Reading the file system source code also enables us to put logic inside our module to interpret blocks as requests pass through it. Lastly, we use offline tools such as `dumpe2fs`, `btrfs-inspect`, and `dump.f2fs` to inspect changes to disk contents. Through these multiple techniques, we can identify block types and specific data structures within the blocks. Table 3 summarizes our approach to identify different data structures in each of the file systems.

After identifying all the relevant data structures for each program, we re-initialize the disk image and repeat test program execution for error injection experiments. We use the same tools, along with our module, to inject errors to specific targets. A single block I/O error or data corruption is injected into a block or data structure during each execution. This allows us to achieve better isolation and characterization of the file system's reaction to the injected error.

Our error injection experiments allow us to measure both immediate and longer-term effects of device faults. We can observe immediate effects on program execution for some cases, such as user space errors or kernel panics (e.g. from write I/O errors). At the end of each test program execution, we unmount the file system and perform several offline tests to verify the consistency of the disk image, regardless of whether the corruption was silent or not (e.g. persisting lost/shorn writes): we invoke the file system's integrity checker (*fsck*), check if the file system is mountable, and check whether

Symbol	Level	Description
○	$\mathbf{D_{Zero}}$	No detection.
–	$\mathbf{D_{ErrorCode}}$	Check the error code returned from the lower levels.
\	$\mathbf{D_{Sanity}}$	Check for invalid values within the contents of a block.
/	$\mathbf{D_{Redundancy}}$	Checksums, replicas, or any other form of redundancy.
|	$\mathbf{D_{Fsck}}$	Detect error using the system checker.
○	$\mathbf{R_{Zero}}$	No attempt to recover.
/	$\mathbf{R_{Retry}}$	Retry the operation first before returning an error.
|	$\mathbf{R_{Propagate}}$	Error code propagated to the user space.
\	$\mathbf{R_{Previous}}$	File system resumes operation from the state exactly before the operation occurred.
–	$\mathbf{R_{Stop}}$	The operation is terminated (either gracefully or abruptly); the file system may be mounted as read-only.
■ (red)	$\mathbf{R_{Fsck_Fail}}$	Recovery failed, the file system cannot be mounted.
■ (orange)	$\mathbf{R_{Fsck_Partial}}$	The file system is mountable, but it has experienced data loss in addition to operation failure.
■ (yellow)	$\mathbf{R_{Fsck_Orig}}$	Current operation fails, file system restored to pre-operation state.
■ (green)	$\mathbf{R_{Fsck_Full}}$	The file system is fully repaired and its state is the same with the one generated by the execution where the operation succeeded without any errors.

Table 4: *The levels of our detection and recovery taxonomy.*

the program's operations have been successfully persisted by comparing the resultant disk image against the expected one. We also explore longer-term effects of faults where the test programs access data that were previously persisted with errors (read I/O, reading corrupted or shorn write data).

In this study, we use *btrfs-progs v4.4*, *e2fsprogs v1.42.13*, and *f2fs-tools v1.10.0* for our error injection experiments.

2.6 Detection and Recovery Taxonomy

We report the *detection* and *recovery* policies of all three file systems with respect to the data structures involved. We characterize each file system's reaction via all observable interfaces: system call return values, changes to the disk image, log messages, and any side-effects to the system (such as kernel panics). We classify the file system's detection and recovery based on a taxonomy that was inspired by previous work [45], but with some new extensions: unlike [45], we also experiment with file system integrity checkers and their ability to detect and recover from errors that the file system might not be able to deal with and as such, we add a few additional categories within the taxonomy that pertain to file system checkers. Also, we create a separate category for the case where the file system is left in its previous consistent state prior to the execution of the program ($R_{Previous}$). In particular, if the program involves updates on the system's metadata, none of it is reflected to the file system. Table 4 presents our taxonomy in detail.

A file system can detect the injected errors online by checking the return value of the block I/O request ($D_{ErrorCode}$), inspecting the incoming data and performing some sanity checks (D_{Sanity}), or using redundancies, e.g. in the form of checksums ($D_{Redundancy}$). A successful detection should alert the user via system call return values or log messages.

To recover from errors, the file system can take several actions. The most basic action is simply passing along the error code from the block layer ($R_{Propagate}$). The file system can also decide to terminate the execution of the system call, either gracefully via transaction abort, or abruptly such as crashing the kernel (R_{Stop}). Lastly, the file system can perform retries (R_{Retry}) in case the error is transient, or use its redundancy data structures to recover the data.

It is important to note that for block I/O errors, the actual data stored in the block is not passed to the disk or the file system. Hence, no sanity check can be performed and D_{Sanity} is not applicable. Similarly, for silent data corruption experiments, our module does not return an error code, so $D_{ErrorCode}$ is not relevant.

We also run each file system's *fsck* utility and report on its ability to detect and recover file systems errors offline, as it may employ different detection and recovery strategies than the online file system. The different categories for *fsck* recovery are shown in Table 4.

3 Results

Tables 5 and 6 provide a high-level summary of the results from our error injection experiments following the detection and recovery taxonomy from Table 4. Our results are organized into six columns corresponding to the fault modes we emulate. The six tables in each column represent the fault detection and recovery results for each file system under a particular fault. The columns (a-w) in each table correspond to the programs listed in Table 2, which specify the operation during which the fault mode was encountered, and rows correspond to the file system specific data structure, that was affected by the fault.

Note that the columns in Tables 5 and 6 have a one-to-one correspondence to the fault modes described in Section 2 (Table 1), with the exception of *shorn writes*. After a shorn write is injected during test program execution and persisted to the flash device, we examine two scenarios where the persisted partial data is accessed again: during fsck invocation (*Shorn Write + Fsck* column) and test program execution (*Shorn Write + Program Read* column).

3.1 Btrfs

We observe in Table 5 that Btrfs is the only file system that consistently detects all I/O errors as well as corruption events, including those affecting data (rather than only metadata). It achieves this through the extensive use of checksums.

However, we find that Btrfs is much less successful in recovering from any issues than the other two file systems. It is the only file system where four of the six error modes can lead to a kernel crash or panic and subsequently a file system that cannot be mounted even after running `btrfsck`. It also has the largest number of scenarios that result in an unmountable file system after `btrfsck` (even if not preceded by a kernel crash). Furthermore, we find that node level checksums, although good for detecting block corruption, they remove an entire node even if a single byte becomes corrupted. As a result, large chunks of data are removed, causing data loss.

Before we describe the results in more detail below, we provide a brief summary of Btrfs data structures. The Btrfs

Btrfs Detection — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+
cksum tree							+										+			+		
root tree																				+		
superblock																				+		
extent tree	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	+		⦶
chunk tree																				+		
dev tree																				+		
uuid tree																				+		
log tree																						
data							+															

Btrfs Detection — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree		+		+	+	+	+	+			+	+	+		+	+	+	+	+		+	
cksum tree																						
root tree	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
superblock	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
extent tree		+		+	+	+	+	+	+	+	+	+	+		+	+		+	+		+	
chunk tree																						
dev tree				+	+		+															
uuid tree																						
log tree		+		+	+		+				+	+			+	+	+		+			
data																						

Btrfs Detection — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ		\ǀ
cksum tree							\ǀ										\ǀ			\ǀ		
root tree																				\ǀ		
superblock																				\ǀ		
extent tree	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ		\ǀ
chunk tree																				\ǀ		
dev tree																				\ǀ		
uuid tree																				\ǀ		
log tree																						
data							\															

Btrfs Recovery — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	⊕	+	+	+		+
cksum tree							ǀ										+			−		
root tree																				−		
superblock																				−		
extent tree	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	+		○
chunk tree																				−		
dev tree																				−		
uuid tree																				−		
log tree																						
data							+															

Btrfs Recovery — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree		−		−	−	○	−	−			−	○			\−	−	\	−	−		−	
cksum tree																						
root tree	−	−	−	−	−	○	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
superblock	○	ǀ	○	\ǀ	\ǀ	\−	−	○	○	○	−	○	−	−	\−	−	\	−	\ǀ−	○	−	
extent tree		○		−	○		−				−	−	−		−	−		−	−		○	
chunk tree																						
dev tree				−	○		−															
uuid tree																						
log tree		ǀ		+	+		+				+	+			+	+	+		+			
data																						

Btrfs Recovery — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
fs tree	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		○
cksum tree							○										+			+		
root tree																				+		
superblock																				+		
extent tree	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	+		○
chunk tree																				+		
dev tree																				+		
uuid tree																				+		
log tree																						
data							+															

ext4 Detection — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				+		
inode	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕		⊕
group desc																				+		
block bitmap		+										+	+		+	+	+					
inode bitmap												+	+		+	+	+		+			
directory	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
extent	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
journal																				+		
data							○															

ext4 Detection — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock		⦶	⦶				⦶					⦶	⦶		⦶		⦶			○	+	
inode		+		+	+	+	+	+		+	+	+	+		+	+	+	+	+		+	
group desc		+										+	+		+	+	+				+	
block bitmap		+										+	+		+	+	+				+	
inode bitmap												+	+		+	+					+	
directory								+				+	+		+	+					+	
extent																	+					
journal		+	+	+	+	+	+	+		+	+	+	+		+	+	+	+	+		+	
data																	−					

ext4 Detection — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				\ǀ		
inode	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	○	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ		\ǀ
group desc																				\ǀ		
block bitmap		\ǀ										\ǀ	\ǀ		\ǀ	\ǀ	\ǀ					
inode bitmap												\ǀ	○		\ǀ	\ǀ						
directory	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ			\ǀ
extent	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ	\ǀ			\ǀ
journal																				\ǀ		
data							○															

ext4 Recovery — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				−		
inode	+	+	+	⊛	⊛	+	+	+	+	+	+	+	+	+	+	+	+	+	+	−		+
group desc																				−		
block bitmap		○										○	○		○	○	○					
inode bitmap												○	○		○	○	○		○			
directory	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
extent	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
journal																				−		
data							+															

ext4 Recovery — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock		○	○				○					○	○		○		○			○	+	
inode		+		+	+	+	+	+		+	+	+	+		+	+	+	+	+		+	
group desc		○										○	○		○	○	○				○	
block bitmap		−										−	−		−	−	−				−	
inode bitmap												−	−		−	−					−	
directory								+				+	+		+	+					+	
extent																	+					
journal		*	*	*	*	*	*	*		*	*	*	*		*	*	*	*	*		+	
data																	○					

ext4 Recovery — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				+		
inode	+	+	+	*	*	+	+	+	+	+	+	⊘	+	+	+	+	+	+	+	+		+
group desc																				+		
block bitmap		○										○	○		○	○	○					
inode bitmap												○	○		○	○						
directory	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
extent	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+
journal																				+		
data							⊘															

F2FS Detection — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				−		
checkpoint																				−		
NAT																				−		
SIT																				−		
inode	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−		−
(d/ind) node							−															
dir. entry	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−			−
data							−															

F2FS Detection — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																						
checkpoint								○				○	○		○	○			○		○	
NAT																						
SIT																						
inode		⦶		⦶	⦶	⦶	⦶	⦶				⦶	⦶		⦶	⦶	⦶		⦶		⦶	
(d/ind) node		⦶														⦶	⦶		⦶		⦶	
dir. entry		⦶		⦶	⦶	⦶	⦶	⦶				⦶	⦶		⦶	⦶	⦶		⦶		⦶	
data																	○				○	

F2FS Detection — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				⦸		
checkpoint																				/		
NAT																				⦸		
SIT																				⦸		
inode	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ	/ǀ		/ǀ
(d/ind) node							⦶															
dir. entry	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶	⦶			⦶
data							○															

F2FS Recovery — Read I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				\		
checkpoint																				\		
NAT																				+		
SIT																				+		
inode	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+
(d/ind) node							+															
dir. entry	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*			*
data							+															

F2FS Recovery — Write I/O Error

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																						
checkpoint								\				\	\		\	\			\		\	
NAT																						
SIT																						
inode		○		○	○	○	○	○				○	○		○	○	○		○		○	
(d/ind) node		○														○	○		○		○	
dir. entry		○		○	○	○	○	○				○	○		○	○	○		○		○	
data																	○				○	

F2FS Recovery — Corruption

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	v	w
superblock																				○		
checkpoint																				○		
NAT																				○		
SIT																				○		
inode	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○
(d/ind) node							○															
dir. entry	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			○
data							○															

Table 5: *The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different read, write, and corruption experiments. The programs that were used are:* ***a:*** *access* ***b:*** *truncate* ***c:*** *open* ***d:*** *chmod* ***e:*** *chown* ***f:*** *utimes* ***g:*** *read* ***h:*** *rename* ***i:*** *stat* ***j:*** *lstat* ***k:*** *readlink* ***l:*** *symlink* ***m:*** *unlink* ***n:*** *chdir* ***o:*** *rmdir* ***p:*** *mkdir* ***q:*** *write* ***r:*** *getdirentries* ***s:*** *creat* ***t:*** *mount* ***v:*** *umount* ***w:*** *chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

○ D_{Zero} − $D_{ErrorCode}$ \ D_{Sanity} / $D_{Redundancy}$ ǀ D_{Fsck}

○ R_{Zero} / R_{Retry} ǀ $R_{Propagate}$ \ $R_{Previous}$ − R_{Stop}

■ R_{Fsck_Full} ■ R_{Fsck_Orig} ■ $R_{Fsck_Partial}$ ■ R_{Fsck_Fail} ■ *Crash/Panic*+R_{Fsck_fail}

Table 6: *The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different shorn write + program read, shorn write + fsck, and lost write experiments. The programs that were used are:* ***a:*** *access* ***b:*** *truncate* ***c:*** *open* ***d:*** *chmod* ***e:*** *chown* ***f:*** *utimes* ***g:*** *read* ***h:*** *rename* ***i:*** *stat* ***j:*** *lstat* ***k:*** *readlink* ***l:*** *symlink* ***m:*** *unlink* ***n:*** *chdir* ***o:*** *rmdir* ***p:*** *mkdir* ***q:*** *write* ***r:*** *getdirentries* ***s:*** *creat* ***t:*** *mount* ***v:*** *umount* ***w:*** *chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

◯ D_{Zero}	– $D_{ErrorCode}$	\ D_{Sanity}	/ $D_{Redundancy}$	I D_{Fsck}
◯ R_{Zero}	/ R_{Retry}	I $R_{Propagate}$	\ $R_{Previous}$	– R_{Stop}
■ R_{Fsck_Full}	■ R_{Fsck_Orig}	■ $R_{Fsck_Partial}$	■ R_{Fsck_Fail}	■ Crash/Panic+R_{Fsck_fail}

file system arranges data in the form of a forest of trees, each serving a specific purpose (e.g. file system tree (*fstree*) stores file system metadata, checksum tree (*csumtree*) stores file/directory checksums). Btrfs maintains checksums for all metadata within tree nodes. Checksum for data is computed and stored separately in a checksum tree. A *root tree* stores the location of the root of all other trees in the file system. Since Btrfs is a copy-on-write file system, all changes made on a tree node are first written to a different location on disk. The location of the new tree nodes are then propagated across the internal nodes up to the root of the file system trees. Finally, the *root tree* needs to be updated with the location of the other changed file system trees.

3.1.1 Read errors

All errors get detected ($D_{\text{ErrorCode}}$) and registered in the operating system's message *log*, and the current operation is terminated (R_{Stop}). `btrfsck` is able to run, detect and correct the file system in most cases, with two exceptions. When the *fstree* structure is affected, `btrfsck` removes blocks that are not readable and returns an I/O error. Another exception is when a read I/O error is encountered while accessing key tree structures during a `mount` procedure: `mount` fails and `btrfsck` is unable to repair the file system.

3.1.2 Corruption

Corruption of any B-tree node. Checksums inside each tree node enable reliable detection of corruption; however, Btrfs employs a different recovery protocol based on the type of the underlying device. When Btrfs is deployed on top of a hard disk, it provides recovery from metadata corruption using metadata replication. Specifically, reading a corrupted block leads to `btrfs-scrub` being invoked, which replaces the corrupted primary metadata block with its replica. Note that `btrfs-scrub` does not have to scan the entire file system; only the replica is read to restore the corrupted block. However, in case the underlying device is an SSD, Btrfs turns off metadata replication by default for two reasons [7]. First, an SSD can remap a primary block and its replica internally to a single physical location, thus deduplicating them. Second, SSD controllers may put data written together in a short time span into the same physical storage unit (i.e. cell, erase block, etc.), which is also a unit of SSD failures. Therefore, `btrfs-scrub` is never invoked in the case of SSDs, as there is no metadata duplication. This design choice causes a single bit flip of *fstree* to wipe out files and entire directories within a *B-Tree* node. If a corrupted tree node is encountered while `mount` reads in all metadata trees into memory, the consequences are even more severe: the operation fails and the disk is left in an inconsistent and irreparable state, even after running `btrfsck`.

Directory corruption. We observe that when a node corruption affects a directory, the corruption could actually be recovered, but Btrfs fails to do so. For performance reasons, Btrfs maintains two independent data structures for a directory (`DIR_ITEM` and `DIR_INDEX`). If one of these two becomes corrupted, the other data structure is not used to restore the directory. This is surprising considering that the existing redundancy could easily be leveraged for increased reliability.

3.1.3 Write errors

Superblock & Write I/O errors: Btrfs has multiple copies of its superblock, but interestingly, the recovery policy upon a write error is not identical for all copies. The superblocks are located at fixed locations on the disk and are updated at every write operation. The replicas are kept consistent, which differs from ext4's behavior. We observe that a write I/O error while updating the primary superblock is considered severe; the operation is aborted and the file system remounts as *read-only*. On the other hand, write I/O errors for the secondary copies of the superblock are detected, but the overall operation completes successfully, and the secondary copy is not updated. While this allows the file system to continue writing, this is a violation of the implicit consistency guarantee between all superblocks, which may lead to problems in the future, as the system operates with a reduced number of superblock copies.

Tree Node & Write I/O errors: A write I/O error on a tree node is registered in the operating system's message *log*, but due to the file system's asynchronous nature, errors cannot be directly propagated back to the user application that wrote the data. In almost all cases, the file system is forced to mount as *read-only* (R_{Stop}). A subsequent `unmount` and `btrfsck` run in repair mode makes the device unreadable for *extentTree*, *logTree*, *rootTree* and the root node of the *fstree*.

3.1.4 Shorn Write + Program Read

We observe that the behavior of Btrfs during a read of a shorn block is similar to the one we observed earlier for corruption. The only exception is the superblock, as its size is smaller than the 3/8th of the block and it does not get affected.

3.1.5 Shorn Write + Fsck

Shorn writes on *root tree* cause the file system to become unmountable and unrecoverable even after a `btrfsck` operation. We also find kernel panics during shorn writes as described in Section 3.1.7.

3.1.6 Lost Writes

Errors get detected only during `btrfsck`. They do not get detected or propagated to user space during normal operation. `btrfsck` is unable to recover the file system, which is rendered unmountable due to corruption. The only recoverable case is a lost write to the superblock; for the remaining data structures, the file system eventually becomes unmountable.

3.1.7 Bugs found/reported.

We submitted 2 bug reports for Btrfs. The first bug report is related to the corruption of a `DIR_INDEX` key. The file system was able to detect the corruption but deadlocks while listing

the directory entries. This bug was fixed in a later version [1]. The second bug is related to read I/O errors specifically on the root directory, which can cause a kernel panic for certain programs. We encountered 2 additional bugs during a shorn write that result in a kernel panic, both having the same root cause. The first case involves a shorn write to the root of the *fstree*, while the second case involves a shorn write to the root of the *extent tree*. In both cases, there is a mismatch in the leaf node size, which forces Btrfs to print the entire tree in the operating system's message *log*. While printing the leaf block, another kernel panic occurs where the size of a Btrfs *item* does not match the Btrfs *header*. Rebooting the kernel and running `btrfsck` fails to recover the file system.

3.2 ext4

ext4 is the default file system for many widely used Linux distributions and Android devices. It maintains a *journal* where all metadata updates are sequentially written before the main file system is updated. First, data corresponding to a file system operation is written to the in-place location of the file system. Next, a transaction log is written on the *journal*. Once the transaction log is written on the *journal*, the transaction is said to be *committed*. When the *journal* is full or sufficient time has elapsed, a *checkpoint* operation takes place that writes the in-memory metadata buffers to the in-place metadata location on the disk. On the event of a crash before the transaction is committed, the file system transaction is discarded. If the commit has taken place successfully on the *journal* but the transaction has not been checkpointed, the file system replays the *journal* during remount, where all metadata updates that were committed on the *journal* are recovered from the journal and written to the main file system.

ext4 is able to recover from an impressively large range of fault scenarios. Unlike Btrfs, it makes little use of checksums unless *metadata_csum* feature is enabled explicitly during file system creation. Further, it deploys a very rich set of sanity checks when reading data structures such as directories, inodes and extents[1], which helps it deal with corruptions.

It is also the only one of the three file systems that is able to recover lost writes of multiple data structures, due to its in-place nature of writes and a robust file system checker. However, there are a few exceptions where the corresponding issue remains uncorrectable (see the red cells in Table 6 associated with ext4's recovery).

Furthermore, we observe instances of data loss caused by shorn and lost writes involving write programs, such as `create` and `rmdir`. For shorn writes, ext4 may incur silent errors, and not notify the user about the errors.

Before describing some specific issues below, we point out that our ext4 results are very different from those reported for ext3 in [45], where a large number of corruption events and several read and write I/O errors were not detected or handled properly. Clearly, in the 13 years that have passed since then, ext developers have made improvements in reliability a priority, potentially motivated by the findings in [45].

I/O errors, corruption and shorn writes of Inodes: The most common scenario leading to data loss (but still a consistent file system) is a fault, in particular read I/O error, corruption or shorn write, that affects an inode, which results in the data of all files having their inode structure stored inside the affected inode block becoming inaccessible.

Read I/O errors, corruption and shorn writes of Directory blocks: A shorn write involving a directory block is detected by the file system and eventually, the corresponding files and directories are removed. Empty files are completely removed by *e2fsck* by default, while non-empty files are placed into the *lost+found* directory. However, the parent-child relationship is lost, which we encode as $R_{Fsck_Partial}$.

Write I/O errors and group descriptors: There is only one scenario where `e2fsck` does not achieve at least partial success: When `e2fsck` is invoked after a write error on a group descriptor, it tries to rebuild the group descriptor and write it to the same on-disk location. However, as it is the same on-disk location that generated the initial error, `e2fsck` encounters the same write error and keeps restarting, running into an infinite loop for 3 cases (see R_{Fsck_Fail}).

Read I/O error during mount: ext4 fails to complete the `mount` operation if a read I/O error occurs while reading a critical metadata structure. In this case, the file system cannot be mounted even after invoking `e2fsck`. We observe similar behavior for the other two file systems as well.

3.3 F2FS

F2FS is a log-structured file system designed for devices that use an FTL. Data and metadata are separately written into 6 active *logs* (default configuration), which are grouped by data/metadata, files/directories, and other heuristics. This *multi-head logging* allows similar blocks to be placed together and increases the number of sequential write operations.

F2FS divides its logical address space into the *metadata region* and the *main area*. The metadata region is stored at a fixed location and includes the Checkpoint (CP), the Segment Information Table (SIT), and the Node Address Table (NAT). The checkpoint stores information about the state of the file system and is used to recover from system crashes. SIT maintains information on the *segments* (the unit at which F2FS allocates storage blocks) in the main area, while NAT contains the block addresses of the file system's *nodes*, which comprise of file/directory inodes, direct, and indirect nodes. F2FS uses a two-location approach for these three structures. In particular, one of the two copies is "active" and used to initialize the file system's state during `mount`, while the other is a shadow copy that gets updated during the file system's execution. Finally, each copy of the checkpoint points to its

[1] We report failure results for both directory and file extents together. Since our pre-workload generation creates a number of files and directories in the root directory, at least 1 extent block corresponding to the root directory gets accessed by all programs.

corresponding copy of SIT and NAT.

F2FS's behavior when encountering read/write errors or corruption differs significantly from that of ext4 and Btrfs. While read failures are detected and appropriately propagated in nearly all scenarios, we observe that F2FS consistently fails to detect and report any write errors, independently of the operation that encounters them and the affected data structure. Furthermore, our results indicate that F2FS is not able to deal with lost and shorn writes effectively and eventually suffers from data loss. In some cases, a run of the file system's checker (called `fsck.f2fs`) can bring the system to a consistent state, but in other cases, the consequences are severe. We describe some of these cases in more detail below.

3.3.1 Read errors

Checkpoint / NAT / SIT blocks & read errors. During its `mount` operation, if F2FS encounters a read I/O error while trying to fetch any of the checkpoint, NAT, or SIT blocks, then it mounts as *read-only*. Additionally, F2FS cannot be mounted if the inode associated with the root directory cannot be accessed. In general, `fsck.f2fs` cannot help the file system recover from the error since it terminates with an assertion error every time it cannot read a block from the disk.

3.3.2 Write errors & Lost Writes

We observe that F2FS does not detect write errors (as injected by our framework), leading to different issues, such as corruption, reading garbage values, and potentially data loss. As a result, during our experiments, newly created or previously existing entries have been completely discarded from the system, applications have received garbage values, and finally, the file system has experienced data loss due to an incomplete recovery protocol (i.e. when `fsck.f2fs` is invoked). Considering that F2FS does not detect write I/O errors, lost writes end up having the same effect, since the difference between the two is that a lost write is silent (i.e. no error is returned).

3.3.3 Corruption

Data corruption is reliably detected only for inodes and checkpoints, which are the only data structures protected by checksums, but even for those two data structures, recovery can be incomplete, resulting in the loss of files (and the data stored in them). The corruption of other data structures can lead to even more severe consequences. For example, the corruption of the superblock can go undetected and lead to an unmountable file system, even if the second copy of the superblock is intact. We have filed two bug reports related to the issues we have identified and one has already resulted in a fix. Below we describe some of the issues around corruption in more detail.

Inode block corruption. Inodes are one of only two F2FS data structures that are protected by checksums, yet their corruption can still create severe problems. One such scenario arises when the information stored in the *footer* section of an inode block is corrupted. In this case, `fsck.f2fs` will discard the entry without even attempting to create an entry in the *lost_found* directory, resulting in data loss.

Another scenario is when an inode associated with a directory is corrupted. Then all the regular files stored inside that directory and its sub-directories are recursively marked as *unreachable* by `fsck.f2fs` and are eventually moved to the *lost_found* directory (provided that their inode is valid). However, we observe that `fsck.f2fs` does not attempt to recreate the structure of sub-directories. It simply creates an entry in the *lost_found* directory for regular files in the sub-directory tree, not sub-directories. As a result, if there are different files with the same name (stored in different paths of the original hierarchy), then only one is maintained at the end of the recovery procedure.

Checkpoint corruption. Checkpoints are the other data structure, besides inodes, that is protected by checksums. We observe that issues only arise if both copies of a checkpoint become corrupted, in which case the file system cannot be mounted. Otherwise, the uncorrupted copy will be used during the system's `mount` operation.

Superblock corruption. While there are two copies of the superblock, the detection of corruption to the superblock relies completely on a set of sanity checks performed on (most) of its fields, rather than checksums or comparison of the two copies. If sanity checks identify an invalid value, then the backup copy is used for recovery. However, our results show that the sanity checks are not capable of detecting all invalid values and thus, depending on the corrupted field, the reliability of the file system can suffer.

One particularly dangerous situation is a corruption of the *offset* field, which is used to calculate a checkpoint's starting address inside the corresponding *segment*, as it causes the file system to boot from an invalid checkpoint location during a `mount` operation and to eventually hang during its `unmount` operation. We filed a bug report which has resulted in a new patch that fixes this problem during the operation of `fsck.f2fs`; specifically, the patch uses the (checksum-protected) checkpoint of the system to restore the correct value. Future releases of F2FS will likely include a patch that enables checksums for the superblock.

Another problem with superblock corruption, albeit less severe, arises when the field containing the counter of supported file extensions, which F2FS uses to identify *cold data*, is corrupted. The corruption goes undetected and as a result, the corresponding file extensions are not treated as expected. This might lead to file system performance problems, but should not affect reliability or consistency.

SIT corruption. SIT blocks are not protected against corruption through any form of redundancy. We find cases where the corruption of these blocks severely compromises the consistency of the file system. For instance, we were able to corrupt a SIT block's bitmap (which keeps track of the allocated blocks inside a *segment*) in such a way that the file system hit a bug during its `mount` operation and eventually,

became unmountable.

NAT corruption. This data structure is not protected against corruption and we observe several problems this can create. First, the node ID of an entry can be corrupted and thus, point to another entry inside the file system, to an invalid entry or a non-existing one. Second, the block address of an entry can be corrupted and thus, point to another entry in the system or an invalid location. In both cases, the original entry is eventually marked as *unreachable* by `fsck.f2fs`, since the reference to it is no longer available inside the NAT copy, and placed in the *lost_found* directory. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

Direct/Indirect Node corruption. These blocks are used to access the data of large files and also, the entries of large directories (with multiple entries). Direct nodes contain entries that point to on-disk blocks, while indirect nodes contain entries that point to direct nodes. Neither single nor double indirect nodes are protected against corruption. We observe that corruption of these nodes is not detected by the file system. Even when an invocation of `fsck.f2fs` detects the corruption problems can arise. For example, we find a case where after the invocation of `fsck.f2fs` the system kept reporting the wrong (corrupted) size for a file. As a result, when we tried to create a copy of the file, we received a different content.

Directory entry corruption. Directory entries are stored and organized into blocks. Currently, there is no mechanism to detect corruption of such a block and we observe numerous problems this can create. For example, when the field in a directory entry that contains the length of the entry's name is corrupted the file system returns garbage information when we try to get the file's status. Moreover, the field containing the node ID of the corresponding inode can be corrupted and as a result point to any node currently stored in the system. Finally, an entry can "disappear" by storing a zero value into the corresponding index inside the directory's bitmap.

In the last two cases, any affected entry is eventually marked as *unreachable* by `fsck.f2fs`, since their parent directory does no longer point to it. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

3.3.4 Shorn Write + Program Read

The results when a program reads a block previously affected by a shorn write are similar to those for corruption, since shorn writes can be viewed as a special type of corruption. The only exception is the superblock, as it is a small data structure that happened not to be affected by our experiments.

3.3.5 Shorn Write + Fsck

Directory entries and shorn writes. Blocks that contain directory entries are not protected against corruption. Therefore, a shorn write goes undetected and can cause several problems. First, valid entries of the system "disappear" after invoking `fsck.f2fs`, including the special entries that point to the current directory and its parent. Second, in some cases, we additionally observed that after re-mounting the file system, an attempt to list the contents of a directory resulted in an infinite loop. In both cases, the affected entries were eventually marked as *unreachable* by `fsck.f2fs` and were dumped into the *lost_found* directory. As we have already mentioned, files with identical names in different parts of the directory tree conflict with each other and eventually, only one makes it to the *lost_found* directory. In some cases, `fsck.f2fs` is not capable of detecting the entire damage a shorn write has caused; we ran into a case where after remounting the file system, all the entries inside a directory ended up having the same name, eventually becoming completely inaccessible.

3.3.6 Bugs found/reported

We have filed two bug reports related to the issues we have identified around handling corrupted data and one has already resulted in a fix [2, 4]. Moreover, we have reported F2FS's failure to handle write I/O errors [3].

4 Related Work

Our work is closest in spirit to the pioneering work by Prabhakaran et *al.* [45], however our focus is very different. While [45] was focused on HDDs, we are specifically interested in SSD-based systems and as such consider file systems with features that are attractive for usage with SSDs, including log-structured and copy-on-write systems. None of the file systems in our study existed at the time [45] was written and they mark a significant departure in terms of design principles compared to the systems in [45]. Also, since we are focused on SSDs, we specifically consider reliability issues that arise from the use of SSDs. Additionally, we provide some extensions to the work in [45], such as exploring whether *fsck* is able to detect and recover from those issues that the file systems cannot handle during their online operation.

Gunawi et *al.* [28] make use of static analysis and explore how file systems and storage device drivers propagate error codes. Their results indicate that write errors are neglected more often than read errors. Our study confirms that write errors are still not handled properly in some file systems, especially when lost and shorn writes are considered. In [51], the authors conduct a performance evaluation on the transaction processing system between ext2 and NILFS2. In [42], the authors explore how existing file systems developed for different operating systems behave with respect to features such as crash resilience and performance. Nonetheless, the provided experimental results only present the performance of read and write operations. Recently, two new studies presented their reliability analysis of file systems in a context other than the local file system. Ganesan et *al.* [25] present their analysis on how modern *distributed storage systems* behave in the presence of file-system faults. Cao et *al.* [20] present their study on the reliability of *high-performance parallel systems*.

In contrast, in our work, we focus on local file systems and explore the effect of SSD related errors.

Finally, different techniques involving hardware or modifications inside the FTL have been proposed to mitigate existing errors inside SSDs [14, 15, 22, 35–37, 44].

5 Implications

• ext4 has significantly improved over ext3 in both detection and recovery from data corruption and I/O injection errors. Our extensive test suite generates only minor errors or data losses in the file system, in stark contrast with [45], where ext3 was reported to silently discard write errors.
• On the other hand, Btrfs, which is a production grade file system with advanced features like snapshot and cloning, has good failure detection mechanisms, but is unable to recover from errors that affect its key data structures, partially due to disabling metadata replication when deployed on SSDs.
• F2FS has the weakest detection against the various errors our framework emulates. We observe that F2FS consistently fails to detect and report any write errors, regardless of the operation that encounters them and the affected data structure. It also does not detect many corruption scenarios. The result can be as severe as data loss or even an unmountable file system. We have filed 3 bug reports; 1 has already been fixed and the other 2 are currently under development.
• File systems do not always make use of the existing redundancy. For example, Btrfs maintains two independent data structures for each directory entry for enhanced performance, but upon failure of one, does not use the other for recovery.
• We notice potentially fatal omissions in error detection and recovery for all file systems except for ext4. This is concerning since technology trends, such as continually growing SSD drive capacities and increasing densities as QLC drives which are coming on the market, all seem to point towards increasing rather than decreasing SSD error rates in the future. In particular for flash-focused file systems, such as F2FS, where for a long time focus has been on performance optimization, an emphasis on reliability is needed if they want to be a serious contender for ext4.
• File systems should make every effort to verify the correctness of metadata through sanity checks, especially when the metadata is not protected by a mechanism, such as checksums. The most mature file system in our study, ext4, does a significantly more thorough job at sanity checks than for example F2FS, which has room for improvement. There have also been recent efforts towards this direction in the context of a popular enterprise file system [32].
• Checksums can be a double-edged sword. While they help increase error detection, coarse granularity checksums can lead to severe data loss. For instance, manipulation of even 1 byte of the checksummed file system unit leads to discard of the entire file system unit in the case for Btrfs. Ideally having a directory or a file system level checksum that discards only 1 entity instead of all co-located files/directories should be implemented. A step in this direction is File-Level Integrity proposed for Android [5, 24]. The tradeoff of adding fine-grained checksums is the space and performance overhead, since a checksum protects a single inode instead of a block of inodes. Finally, note that checksums can only help with detecting corruption, but not with recovery (ideally a file system can both detect corruption and recover from it). These points have to be considered together when implementing checksums inside the file system.
• One might wonder whether added redundancy as described in the Iron file system paper [45] might resolve many of the issues we observe. We hypothesize that for flash-based systems, redundancy can be less effective in those (less likely) cases where both the primary and replica blocks land in the same fault domain (same erase block or same flash chip), after being written together within a short time interval. Even though modern flash devices keep multiple erase blocks open and stripe incoming writes among them for throughput, this does not preclude the scenario where both the primary and replica blocks land in the same fault domain.
• Maybe not surprisingly, a few key data structures (e.g. the journal's *superblock* in ext4, the root directory *inode* in ext4 and F2FS, the root node of *fstree* in Btrfs) are responsible for the most severe failures, usually when affected by a silent fault (e.g. silent corruption or silently dropped write).It might be worthwhile to perform a series of sanity checks for such key data structures before persisting them to the SSD e.g. during an `unmount` operation.

6 Limitations and Future Work

Some of the fault types we explore in our study are based on SSD models that are several years old by now, whose internal behavior could have changed since then. However, we observe that some issues are inherent to flash and therefore likely to persist in new generations of drives, such as retention and disturb errors, which will manifest as read errors at the file system level. The manifestation of other faults, e.g. those related to firmware bugs or changes in page and block size, might vary for future drive models. Our tool is configurable and can be extended to test new error patterns.

File systems must remain consistent in the face of different types of faults. As part of future work, we plan to extend our device mapper module to emulate additional fault modes, such as timeouts. Additionally, our work can be expanded to include additional file systems, such as XFS, NTFS and ZFS. Finally, another extension to our work could be exploring how file systems respond to timing anomalies as those described in [26], where I/Os related to some blocks can become slower, or the whole drive is slow.

Acknowledgements

We thank our USENIX ATC '19 reviewers and our shepherd, Theodore Ts'o, for their detailed feedback and valuable suggestions.

References

[1] Btrfs Bug Report. https://bugzilla.kernel.org/show_bug.cgi?id=198457.

[2] F2FS Bug Report. https://bugzilla.kernel.org/show_bug.cgi?id=200635.

[3] F2FS Bug Report - Write I/O Errors. https://bugzilla.kernel.org/show_bug.cgi?id=200871.

[4] F2FS Patch File. https://sourceforge.net/p/linux-f2fs/mailman/message/36402198/.

[5] fs-verity: File System-Level Integrity Protection. https://www.spinics.net/lists/linux-fsdevel/msg121182.html. [Online; accessed 06-Jan-2019].

[6] Github code repository. https://github.com/uoftsystems/dm-inject.

[7] Btrfs mkfs man page. https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs, 2019. [Online; accessed 06-Jan-2019].

[8] 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 (ATC '08)*, volume 57, 2008.

[9] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[10] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th Annual International Reliability Physics Symposium*, pages 7–20. IEEE, 2002.

[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[12] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 115–128. USENIX Association, 2010.

[13] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st Annual International Reliability Physics Symposium*, pages 127–132. IEEE, 1993.

[14] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[15] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In *23rd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 49–60. IEEE, 2017.

[16] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[17] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015.

[18] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.

[19] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *30th International Conference on Computer Design (ICCD)*, pages 94–101. IEEE, 2012.

[20] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 1–11. ACM, 2018.

[21] Paolo Cappelletti, Roberto Bez, Daniele Cantarelli, and Lorenzo Fratin. Failure mechanisms of Flash cell in program/erase cycling. In *Proceedings of the IEEE International Electron Devices Meeting*, pages 291–294. IEEE, 1994.

[22] 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 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of

Computer Systems (SIGMETRICS '09), pages 181–192, 2009.

[23] Robin Degraeve, F Schuler, Ben Kaczer, Martino Lorenzini, Dirk Wellekens, Paul Hendrickx, Michiel van Duuren, GJM Dormans, Jan Van Houdt, L Haspeslagh, et al. Analytical percolation model for predicting anomalous charge loss in flash memories. *IEEE Transactions on Electron Devices*, 51(9):1392–1400, 2004.

[24] Jake Edge. File-level Integrity. https://lwn.net/Articles/752614/, 2018. [Online; accessed 06-Jan-2019].

[25] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[26] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 24–33, Dec 2009.

[27] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*. USENIX Association, 2012.

[28] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 14:1–14:16, San Jose, CA, 2008.

[29] S Hur, J Lee, M Park, J Choi, K Park, K Kim, and K Kim. Effective program inhibition beyond 90nm NAND flash memories. *Proc. NVSM*, pages 44–45, 2004.

[30] Seok Jin Joo, Hea Jong Yang, Keum Hwan Noh, Hee Gee Lee, Won Sik Woo, Joo Yeop Lee, Min Kyu Lee, Won Yol Choi, Kyoung Pil Hwang, Hyoung Seok Kim, et al. Abnormal disturbance mechanism of sub-100 nm NAND flash memory. *Japanese Journal of Applied Physics*, 45(8R):6210, 2006.

[31] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the 2013 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*, pages 203–216, 2013.

[32] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[33] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[34] Jae-Duk Lee, Chi-Kyung Lee, Myung-Won Lee, Han-Soo Kim, Kyu-Charn Park, and Won-Seong Lee. A new programming disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current. In *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, pages 31–33. IEEE, 2006.

[35] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, page 11, San Jose, CA, 2012. USENIX Association.

[36] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 504–517. IEEE, 2018.

[37] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. *Proceedings of the 2018 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '18)*, 2(3):37:1–37:48, December 2018.

[38] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[39] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, pages 177–190, 2015.

[40] Neal Mielke, Hanmant P Belgal, Albert Fazio, Qingru Meng, and Nick Righos. Recovery Effects in the Distributed Cycling of Flash Memories. In *Proceedings of the 44th Annual International Reliability Physics Symposium*, pages 29–35. IEEE, 2006.

[41] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. Bit error rate in NAND flash memories. In *Proceedings of the 46th Annual International Reliability Physics Symposium*, pages 9–19. IEEE, 2008.

[42] Keshava Munegowda, GT Raju, and Veera Manikandan Raju. Evaluation of file systems for solid state drives. In *Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, pages 342–348, 2014.

[43] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, pages 7:1–7:11, 2016.

[44] Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Thomas Mittelholzer, Evangelos Eleftheriou, Charles Camp, Thomas Griffin, Gary Tressler, and Andrew Walls. Using Adaptive Read Voltage Thresholds to Enhance the Reliability of MLC NAND Flash Memory Systems. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI '14)*, pages 151–156, 2014.

[45] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, 2005.

[46] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, August 2013.

[47] Marco AA Sanvido, Frank R Chu, Anand Kulkarni, and Robert Selinger. NAND flash memory and its role in storage architectures. *Proceedings of the IEEE*, 96(11):1864–1874, 2008.

[48] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[49] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, 1995.

[50] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the Impact of Power Loss on Flash Memory. In *Proceedings of the 48th Design Automation Conference (DAC '11)*, pages 35–40, San Diego, CA, 2011.

[51] Yongkun Wang, Kazuo Goda, Miyuki Nakano, and Masaru Kitsuregawa. Early experience and evaluation of file systems on SSD with database applications. In *5th International Conference on Networking, Architecture, and Storage (NAS)*, pages 467–476. IEEE, 2010.

[52] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs Under Power Fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, CA, 2013. USENIX Association.

[53] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability Analysis of SSDs Under Power Fault. *ACM Transactions on Storage (TOS)*, 34(4):1–28, November 2016.

Alleviating Garbage Collection Interference through Spatial Separation in All Flash Arrays*

Jaeho Kim Kwanghyun Lim[‡] Young-Don Jung[†] Sungjin Lee[†] Changwoo Min Sam H. Noh[⋆]

Virginia Tech [‡]*Cornell University* [†]*DGIST* [⋆]*UNIST*

Abstract

We present SWAN, a novel All Flash Array (AFA) management scheme. Recent flash SSDs provide high I/O bandwidth (e.g., 3-10GB/s) so the storage bandwidth can easily surpass the network bandwidth by aggregating a few SSDs. However, it is still challenging to unlock the full performance of SSDs. The main source of performance degradation is garbage collection (GC). We find that existing AFA designs are susceptible to GC at SSD-level and AFA software-level. In designing SWAN, we aim to alleviate the performance interference caused by GC at both levels. Unlike the commonly-used *temporal separation approach* that performs GC at idle time, we take a *spatial separation approach* that partitions SSDs into the front-end SSDs dedicated to serve write requests and the back-end SSDs where GC is performed. Compared to temporal separation of GC and application I/O, which is hard to be controlled by AFA software, our approach guarantees that the storage bandwidth always matches the full network performance without being interfered by AFA-level GC. Our analytical model confirms this if the size of front-end SSDs and the back-end SSDs are properly configured. We provide extensive evaluations that show SWAN is effective for a variety of workloads.

1 Introduction

With the advent of the IoT and big data era, the amount of data generated, manufactured, and processed is expected to grow at rates that were previously unimaginable [20, 25, 36, 44, 52]. Such explosive growth of data will impose considerable stress on storage systems in data centers. All Flash Array (AFA) storage, which solely uses an array of SSDs, seems to be a viable storage solution that is capable of satisfying such a high demand. AFA has been recently receiving a lot of attention because of its high performance, low power consumption, and high capacity per volume.

While flash SSD is relatively new and its characteristics are different from HDD, the overall architecture of an AFA system is not much different from traditional HDD-based storage servers [9, 18, 21, 30, 35, 58]. This is because, instead of architecting a new SSD-based storage server from scratch, existing HDD-based storage servers have evolved to embrace high-speed SSDs. For example, an array of SSDs inside an AFA are grouped by variants of RAID architectures (e.g., RAID4, RAID5, or Log-RAID, which is based on log-structured writing that we describe in more detail in Section 2).

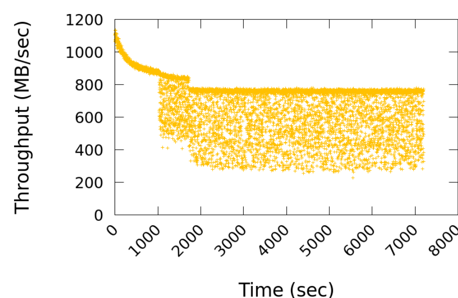


Figure 1: Performance of AFA consisting of eight SSDs under random write workloads. Performance fluctuation starts to occur (at around 1000 seconds) when the size of user write request approaches the capacity of AFA, roughly 1 TB in this configuration.

This naive AFA design, which replaces HDDs with SSDs, is not adequate to take full advantage of high-speed SSDs in two reasons. First, we observe significant performance drop as we run the FIO tool [6] that generates 8 KB random writes. Figure 1 shows the throughput of AFA with eight SATA SSDs (Samsung's 850 PRO) grouped as Log-RAID [9, 10, 21], where each SSD exhibits an effective write throughput of 140 MB/s. We find that garbage collection (GC) of AFA interfere with user I/O. Specifically, for the first 1,000 seconds, the system maintains high throughput that is close to the accumulated throughput of the eight SSDs. However, after 1,000 seconds, owing to interference with GC, the throughput drops considerably and oscillates between 300 MB/s and 750 MB/s, which is much lower than its full performance for 8 KB random I/Os.

*This work was initiated while Jaeho Kim was a postdoc at UNIST.

Table 1: Comparison of All-Flash-Array products

		EMC XtremIO [1]	NetApp SolidFire [4]	HPE 3PAR [2]	Hynix AFA [22]
SSD Array	Capacity	36∼144TB	46TB	750TB	552TB
	# of SSDs	18∼72	12	120 (max)	576
Network	Network Ports	4∼8×10Gb iSCSI	2×25Gb iSCSI	4∼12×16Gb FC	3×Gen3 PCIe
	Aggr. Network Throughput	5∼10 GB/s	6.25 GB/s	8∼24 GB/s	48 GB/s

Second, we find that storage bandwidth and network bandwidth are unbalanced. Typically, AFA is composed of multiple bricks, which is a 1U storage node to scale out capacity. As shown in Table 1, each brick is composed of a large number of SSDs and multiple network ports. Given the storage capacity and the number of installable SSDs, the aggregated write throughput of SSDs easily surpasses the aggregated network throughput. Taking EMC's XtremIO in Table 1 as an example, its 10GB/s network throughput can be easily saturated with four high-end SSDs with 2.5GB/s write throughput [48] out of the 18∼72 SSDs. This matches with observations of other recent work [12, 15, 37, 40].

The above two observations lead us to propose a new architecture for AFA systems. Given the maximum network or user-required throughput, our goal is to derive a balanced AFA system that *satisfies* the required throughput all the time without being interfered with foreground GC. To this end, we present *SWAN*, which stands for Spatial separation Within an Array of SSDs on a Network. In contrast to RAID that manages *all* the SSDs in parallel, SWAN manages the SSDs through several spatially separated groups. That is, only *some* spatially segregated SSDs out of all the SSDs, are in use at a single point in time to serve write requests over the network.

The rationale behind such segregated management is that, even if a large number of SSDs are available in the system, all the SSDs are not necessary to fully saturate the network bandwidth of the AFA. Using more SSDs in parallel does not help the clients in terms of performance. However, even with such a small number of SSDs being used, providing ideal, consistent performance is impossible with GC interference. The only way to achieve such ideal performance is hiding the GC effect. To hide the GC effect in SWAN, we partition the SSDs in the array into two *spatially* exclusive groups, that is, the front-end and the back-end SSDs, of which the front-end is composed of enough SSDs to saturate the bandwidth specification of an AFA. Then, SWAN manages these SSDs such that all writes are sequentially written to the front-end SSDs and those SSDs are never involved in GC. While the front-end SSDs are busy handling the user writes, the back-end SSDs perform GC in the background. Once the front-end SSDs become full with user data, the cleaned back-end SSDs become the new front-end to keep serving the user writes without foreground GC. By so doing, performance interference due to GC can be hidden.

This unique organization and operational behavior of SWAN gives us insight in deriving its balanced design. The number of SSDs belonging to the front-end SSDs should be large enough to fully saturate the required throughput. If not, users' performance demand cannot be satisfied. To give enough time for back-end SSDs to be completely cleaned up before they become frond-end SSDs, we should provision enough SSDs in the front-end and back-end SSD pool. Otherwise, foreground GC becomes unavoidable. The number of SSDs in the frond-end and back-end groups can be estimated by referring to the required throughput and worst-case GC cost models [31, 38, 53].

In summary, this paper makes the following contributions:

- We present a two-dimensional SSD organization as a new AFA architecture, which we refer to as SWAN. We show that such an organization allows for spatial separation of arrays of SSDs so that consistent throughput that is not influenced by GC can be provided.
- We provide an analytic model that decides the best number of SSDs in the frond-end SSD group and in the back-end SSD group. This provides guidance on deriving a balanced system when designing SWAN-based AFAs.
- We conduct comprehensive evaluations using various workloads, including both synthetic and realistic workloads, and demonstrate that SWAN outperforms existing storage management techniques such as RAID0, RAID4, RAID5, and Log-RAID [9, 10, 21]

The remainder of this paper is organized as follows. In the next section, we review existing AFA systems. Then, in Section 3, we present the design of SWAN in detail and an analytic model to completely hide the performance interference by GC. After discussing the implementation issues of SWAN in Section 4, we describe our experimental setup and present detailed results with various workloads in Section 5. We discuss the influence on SWAN on SSD design and other relevant issues in Section 6, . We review prior studies related to this work, comparing them with SWAN in Section 7. Finally, we conclude the paper with a summary in Section 8.

2 Background: All Flash Array

Existing AFA systems can be roughly categorized into two types depending on their write strategies, in-place write AFAs and log write AFAs, as summarized in Table 2.

Table 2: Comparison of existing approaches for managing All-Flash-Array storage

	Write Strategy	GC Interference	Media type	Modification	Disk Organization
Harmonia [30]	In-place write	●	SSDs	Array controller	RAID-0
HPDA [35]	In-place write	●	SSDs & HDDs	Host layer	RAID-4
GC-Steering [58]	In-place write	●	SSDs	Host layer	RAID-4/5
SOFA [9]	Log write	●	SSDs	Host layer	Log-RAID
SALSA [21]	Log write	●	SSDs & SMR	Host layer	Log-RAID
Purity [10]	Log write	●	SSDs	Host layer	Log-RAID
SWAN (proposed)	Log write	▲	SSDs	Host layer	2D Array

GC Interference: Interference between GC of SSD/AFA-level and user's I/O ●: Heavy interference ▲: Alleviated interference

In-place Write AFAs. Approaches with the in-place write strategy heavily rely on the traditional RAID architecture, but attempts have been made to improve performance by modifying data placement and the GC mechanism. The most well-known ones are Harmonia [30], HPDA [35], and GC-Steering [58].

Harmonia is based on the RAID0 architecture that groups all SSDs in an array in parallel without any parity disks [30]. To minimize high I/O fluctuation caused by SSD-level GC that independently happens in individual SSDs, it proposes a globally-coordinated GC algorithm that synchronizes the GC invocations across all the SSDs in the array. If GC is invoked in one SSD, it intentionally triggers GC in all the others, so that all the SSDs in the array are garbage collecting. This approach minimizes user-perceived I/O fluctuation through frequent GC invocations, but cannot completely eliminate performance interference by GC.

HPDA (Hybrid Parity-based Disk Array) takes an SSD-HDD hybrid architecture based on RAID4 [35]. By using a few HDDs as temporary storage to keep parity information, it mitigates performance and lifetime degradations of SSDs caused by frequent updates of parity data. While it is effective in lowering parity overhead, it does not propose any technique to hide GC interference.

GC-Steering is similar to SWAN [58]. It spatially dedicates few SSDs, called staging disks, to absorb the host writes while the other SSDs are busy performing GC. Since host writes can be served by staging disks, GC-Steering can prevent clients from being influenced by GC. However, once the staging disks become full and run out of free space to service host writes, foreground GC becomes unavoidable. To be more specific, it differs from SWAN in that it inherits the limitation of a cache. The staging space is divided into read and write regions, and hot data needs to be migrated to the space for read requests. Space constraints in the staging space not only make it impossible for all reads to avoid collision with GC, but it also causes migration overhead.

Log Write AFAs. While using the traditional RAID architecture (e.g., RAID4 or 5) for the purpose of fault-tolerance, some studies adopt log-structured writing, fundamentally changing its storage management policy, so as to generate sequential write requests that are more suitable for flash-based SSDs. To distinguish them from traditional ones, in this paper, we call AFA systems with log writing a log-structured RAID (Log-RAID).

SOFA is one of the first attempts to use the log-structured approach for AFAs [9]. SOFA integrates volume management, flash translation layer (FTL), and RAID logic together and runs them all in the host level. This integration enables global management of GC and wear-leveling, resulting in overhead associated with storage maintenance tasks being considerably reduced.

SALSA is similar to SOFA in that it offloads almost all of the functions that are typically performed inside storage devices to the host level [21]. However, unlike SOFA, SALSA's primary aim is in building a general-purpose storage platform that supports various types of storage media that are incapable of supporting in-place updates such as SSD and SMR.

Purity is an AFA appliance developed by Pure Storage [10]. Purity adopts log-structured indexes and data layouts that are based on the LSM-tree algorithm [43] to ensure that data is written in large sequential chunks. For better utilization of disk capacity, it also incorporates compression and deduplication algorithms into their system.

SWAN shares the same advantages of the aforementioned techniques as it runs its log-structured storage management logic at the host level. However, SWAN takes it one step further by minimizing the performance interference caused by GC while considering an AFA design that balances storage media and network interface performance.

3 Design of SWAN

3.1 Design Goal and Approach

The primary design goal of SWAN is to provide sustainable high performance for All Flash Array (AFA). More specifically, so that storage does not become the bottleneck, we aim to guarantee AFA storage performance to always be higher than or equal to the network interface bandwidth of AFA.

At a glance, this looks easy to achieve by simply using a RAID of multiple SSDs because even consumer-grade SSDs provide more than 1 GB/s bandwidth. As shown in Figure 2(a), RAID can be used to improve performance and reliability of AFA by composing multiple SSDs in parallel.

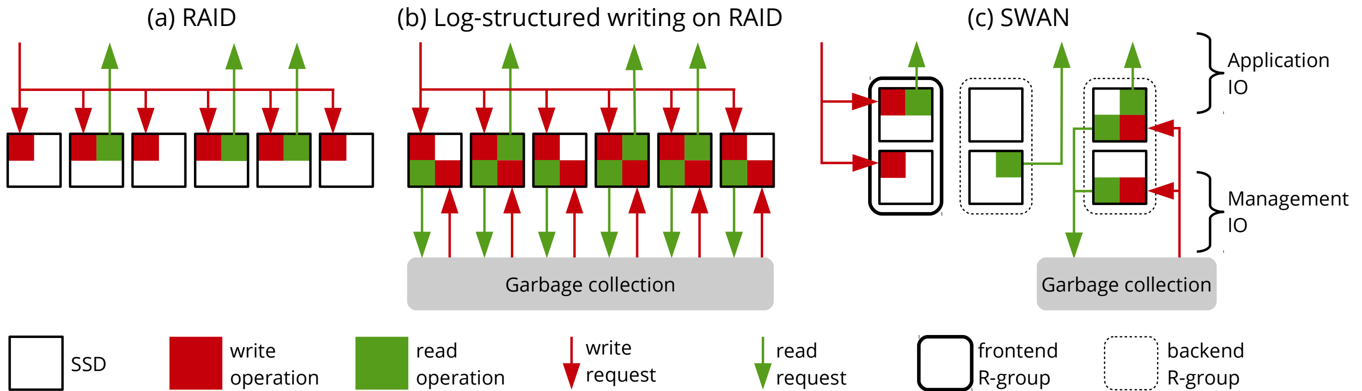


Figure 2: Comparison of All Flash Array (AFA) design. Existing AFA roughly follow either (a) RAID or (b) log-structured writing on RAID (Log-RAID) approaches. We propose (c) SWAN, a spatial separation approach to AFA.

However, its design is susceptible to gradual performance degradation due to high GC overhead inside SSDs. In fact, it turns out that such *SSD-level GC* can significantly degrade performance and incur latency spikes in AFA [30, 51].

To mitigate the internal SSD-level GC overhead, log-structured writing on RAID (Log-RAID), as depicted in Figure 2(b), has been widely adopted in AFA [9, 10, 21]. It generates SSD-friendly write requests by transforming small, random write requests to a bulk, sequential write stream thereby reducing the internal GC overhead in SSDs. However, the I/O operations for *AFA-level GC* (not SSD-level) may significantly interfere with application I/O and degrade performance by constantly issuing read/write requests. In particular, if an application tries to read a sector on an SSD where AFA-level GC is in progress, read latency could increase by several orders of magnitude [21, 23, 51]. A common approach to mitigate such interference is to perform AFA-level GC at idle time by *temporally separating* application I/O and AFA-level GC I/O, that is, segregate the two I/Os in terms of time. However, temporal separation of application I/O and AFA-level GC I/O is hard to control in reality because high performance AFAs are designed to handle multiple concurrent clients.

The key idea of our approach is the *spatial separation* of application I/O and AFA-level GC I/O to minimize such interference by organizing the SSDs into a two-dimensional array as depicted in Figure 2(c). Like Log-RAID, we adopt log-structured writing on RAID to minimize the performance degradation caused by heavy SSD-level GC while providing high performance and reliability using RAID. However, unlike Log-RAID, we spatially separate SSDs into two pools, the front-end and back-end pools. The front-end pool will serve write requests from applications in a log-structured manner, while the back-end pool is used for *SWAN-level GC*, which is GC that occurs only at the back-end pool. Thus, SWAN-level GC does not interfere with application write requests. When the front-end pool SSDs become full, a pool of SSDs from the back-end becomes the front-end and the old front-end SSDs return to the back-end. This design also has the advantage that application read requests are less interfered by SWAN-level GC operations. Recall that as SWAN uses commodity SSDs, it does not have direct control over SSD-level GC. However, we take a best effort approach given the conventional SSD interface.

3.2 Flash Array Organization

SWAN exposes linear 4KB logical blocks such that upper-level software just considers SWAN as a large block device. The SWAN software module is implemented as a part of the Linux kernel in the block I/O layer, where a logical volume manager or a software RAID layer is implemented. SWAN groups multiple SSDs into a single physical volume, which is then divided into fixed-size segments. It manages each segment in a log-structured manner, with new data always being sequentially appended. A segment is the unit for writing a chunk of data as well as for cleaning of obsolete data. Similar to other log-structured systems, therefore, SWAN manages mapping between logical blocks and segments, and, when necessary, it performs GC to secure free segments.

The overall architecture of SWAN is like a big host-level FTL supporting multiple SSDs, but it is the management mechanism in SWAN that differs from typical systems. Typical RAID systems manage an SSD array in a one-dimensional manner, that is, all the SSDs are arranged horizontally and incoming writes are evenly striped over all of them. Unlike RAID, in SWAN, an array of SSDs is organized as a *two-dimensional array*. Then, SSDs belonging to the same column are grouped in a RAID manner and are used in parallel. This group of SSDs is called a *RAID group (R-group)* as this is where redundancy is manifested to prevent data loss in the event of hardware or power failure. The R-group is also where the size can be set such that its aggregate throughput surpasses the network interface provided by AFA. That is, for AFA providing higher network interface bandwidth, we can increase the number of SSDs within the

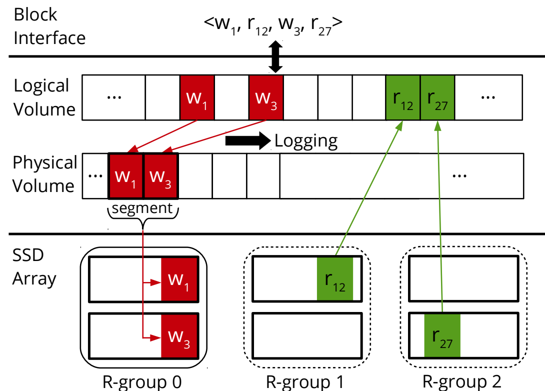


Figure 3: Example of handling read/write requests in SWAN where R-group 0 is the front-end R-group and R-groups 1 and 2 are in the back-end pool. SWAN appends writes to the log and issues write requests to the front-end R-group in segment units. Read requests will be served by any R-group holding the requested blocks.

R-group to match the network bandwidth. We further discuss optimizing the configuration of SWAN in Section 3.5.

3.3 Handling Application I/O Requests

As discussed, SWAN organizes SSDs into two or more R-groups, and each R-group is either a front-end or belongs to the back-end pool at a certain point in time. SWAN manages the movement of the R-groups such that each R-group takes turns being the front-end R-group, while the rest belong to the back-end pool.

Only the front-end R-group serves the incoming writes in a log-structured manner, consuming its free space. Once free space of SSDs in the R-group in the front-end pool is exhausted, SWAN moves this R-group to the back-end pool and selects a new R-group in the back-end pool that has enough free space to become the front-end R-group to serve writes coming in from the network. Incoming read requests are served by any SSD holding the requested blocks.

Figure 3 shows an example of how SWAN handles the I/O sequence $< w_1, r_{12}, w_3, r_{27} >$ arriving from the network, where w_i and r_i are the write and read of block i, respectively. The writes are appended to a segment, but are actually distributed across SSDs in the front-end R-group and are written in parallel. Reads, in contrast, will be served by any of the three R-groups. To alleviate read delays due to GC, we can employ methods such as RAIN as suggested by Yan et al. [60], which we do not consider in this study and leave for future work. However, as we show later, even without such optimizations, SWAN read performance does not suffer from delays as it is always given highest priority. Thus, read performance is comparable with conventional methods, while write performance is significantly improved.

3.4 Garbage Collection in SWAN

SWAN performs GC to secure free segments like other log-structured systems. SWAN chooses victim segments from one of the back-end R-groups and writes valid blocks within the chosen R-group. That is, GC is performed internally within a single back-end R-group. Also, while any victim selection policy could be used [13, 16], in this paper, we use the greedy policy that chooses a segment that has the least number of valid blocks as the victim. Such GC creates a free segment in the chosen R-group. When the front-end R-group becomes full, SWAN chooses an R-group in the back-end to be the next front-end, then moves the old front-end to the back-end group. SWAN completely decouples normal I/O from GC I/O by spatially separating SSDs into the front-end R-group and the back-end R-groups, so as to eliminate interference by GC I/O upon user writes.

3.5 Optimizing SWAN Configuration

The key insight behind the SWAN design is that given the many SSDs in AFA, only a portion of these SSDs (which forms the R-group) are sufficient to saturate the network interface bandwidth of an AFA. However, to realize and achieve sustainable high performance in SWAN, it is important to properly decide the configuration knobs: 1) the number of SSDs in an R-group and 2) the minimum number of R-groups in an SSD array.

Determining the number of SSDs in an R-group. The aggregated throughput of the SSDs in one R-group must be high enough to fully saturate the AFA network interface bandwidth. Thus, we determine the number of SSDs in an R-group such that the aggregated write throughput[1] of the SSDs in an R-group is higher than the aggregated AFA network interface bandwidth. For example, using an AFA configuration such as the EMC XtremIO in Table 1, given the aggregate network throughput of 10 GB/s and assuming the maximum write throughput of an SSD to be 2.5 GB/s [48], four SSDs (three for data and one for parity) must be assigned to the R-group to support RAID4 in SWAN.

Determining the minimum number of R-groups. Besides the raw aggregated throughput of an R-group, another important factor to decide the maximum write throughput of SWAN is GC overhead. If consuming a segment in a front-end R-group is faster than generating a free segment in the back-end R-groups, SWAN-level GC will be a performance bottleneck, limiting its performance. Thus, the number of the back-end R-groups should be large enough for SWAN-level GC not to fall behind. We provide an *analytic model* to calculate the minimum number of R-groups, which determines the number of back-end R-groups, to avoid cases where GC falls behind the front-end writing. In our analytic model, we only consider operations that dominate the execution time

[1] We consider only the write throughput of an SSD because read is faster than write.

such as read, write, and garbage collection and do not consider SSD-level optimizations such as the write buffer in the SSD that could further improve SSD performance.

Since SWAN manages the SSD array in a log-structured fashion, it divides the SSD space into fixed-size segments, and all the write and cleaning operations are done in segment units. Let T_w and T_r denote the elapsed times for writing and reading a segment to and from an SSD, respectively. Let T_e be the time for erasing flash blocks in a segment when all the data in it are invalid.

SWAN writes new data over the network to a front-end R-group. Once the front-end R-group fills up, it is moved to the back-end. To perform GC for a segment that has both valid and invalid pages in the back-end, the valid pages must first be read and then written to a free segment. Of course, all flash blocks in the free segment must be erased before writing valid pages. Therefore, time to finish GC of an R-group is $S \cdot (T_e + u \cdot (T_r + T_w))$ where S is the number of segments in an R-group and u is the ratio of valid pages in a segment (i.e., segment utilization). After finishing GC of an R-group, we will have $S \cdot (1-u)$ free space for this R-group. That R-group will be moved to the front-end later at some time. It will take $S \cdot (1-u) \cdot T_w$ to completely consume the free segments in that R-group.

In SWAN, all R-groups are independent of each other; either they service writes and reads as the one in the front-end or they perform GC and reads as ones in the back-end. Once an R-group is moved to the back-end pool, it will not be chosen as a front-end R-group until all previous R-groups in the back-end are consumed. This implies that, after an R-group moves to the back-end pool, it will return as the front-end R-group after $(N-1) \cdot (S \cdot (1-u) \cdot T_w)$ time at the earliest, where N is the number of R-groups in a SWAN array. This is when data are written to the front-end R-group SSDs at maximum throughput; it will take longer to return if the throughput is lower.

This tells us that, if the GC time of an R-group is equal to or shorter than the time that R-group is recycled, SWAN can finish GC of an R-group before moving it to the front-end. Conversely, if the above condition is not met, SSDs in the front-end R-group may need to delay writes as it waits for free segments to become available.

Consequently, the condition

$$S \cdot (T_e + u \cdot (T_r + T_w)) \leq (N-1) \cdot S \cdot (1-u) \cdot T_w$$

must hold to guarantee that SWAN-level GC does not interfere application writing at the front-end. This can be simplified as follows:

$$T_e + u \cdot (T_r + T_w) \leq (N-1) \cdot (1-u) \cdot T_w$$

From here, we get

$$\frac{T_e}{T_w} \cdot \frac{1}{1-u} + \left(\frac{T_r}{T_w} + 1\right) \cdot \frac{u}{1-u} + 1 \leq N \tag{1}$$

Note that Equation 1 is independent of the number of SSDs per R-group and dependent on the specifications of the SSDs and the utilization. Previous studies [31, 38, 53] have shown that in a log-structured scheme, $u_d = \frac{u-1}{\ln u}$ holds, where u_d is the disk utilization. This tells us that even for heavy loaded storage systems where the disk utilization (u_d) is 60% to 70%, u will be below 0.5.

Let us now consider applying the model. Given an array of SSDs, let T_e, $T_w = t_w \cdot B$, and $T_r = t_r \cdot B$ be constants, where T_e, t_w, t_r, and B are the time to erase a segment, write a block, read a block, and the number of blocks in a segment, respectively. From our AFA prototype, our measurements show that segment erase time is roughly 4 milliseconds, block read and write time is 15.6 and 19.5 microseconds, respectively, and the number of blocks per segment is 256. Taking these numbers and with a storage device that is ($u_d =$) 60% utilized, which results in roughly $u = 0.33$, then we can calculate N to be roughly 1.89. This tells us that with SWAN composed of two R-groups, we will be able to sustain the full network interface bandwidth performance and see no GC affects throughout its services. We believe that the modeling results based on our measurement-based parameter estimations effectively reflects the underlying system architecture as the impact of realistic factors such as queuing delays and resource contention are being reflected in the measured parameters.

Applying these results to a realistic setting, let us, once again, take a configuration such as EMC's XtremIO in Table 1, assuming an SSD with 2.5 GB/s write bandwidth. If we can configure each R-group to be of four SSDs, which is enough to saturate the network bandwidth, then we have, in the smallest configuration case (18 SSDs), three back-end R-groups (plus two spare SSDs), which will be more than sufficient to allow full sustained write performance.

One factor that we did not consider in our analysis is the bandwidth consumed by read requests to the back-end R-group. However, in reality, as the number of back-end R-groups are sufficiently high, these reads will not have a real effect on GC time needed to return as a front-end R-group.

Our analysis shows that with our SWAN approach, once we have set the number of SSDs within the R-group to match the network bandwidth, the total number of SSDs to maintain high, sustained performance can be determined. Also, extra SSDs for larger capacity will further ensure that SWAN-level GC will not interfere user writes at the front-end.

4 Implementation

We implement SWAN and Log-RAID in the block I/O layer, where the I/O requests are redirected from the host to the storage devices, in Linux kernel-3.13.0. For our implementation, we extend the SSD RAID Cache implementation in the Device Mapper (DM) [42] to accommodate AFA storage. To implement RAID0, RAID4, and RAID5, we use `mdadm`, which is a GNU/Linux utility used to manage and monitor

software RAID devices [54].

4.1 Metadata Management

SWAN and Log-RAID maintain basically the same metadata. They manage two types of metadata: 1) a mapping table from the logical volume to the physical volume mapping table (L2P) for address translation, and 2) the segment summary information. Each entry in the table, which takes up 5 bytes, corresponds to a 4 KB block in an SSD array. Thus, the metadata size for the mapping table is roughly 0.12% of the total storage capacity (i.e., 5 bytes per 4 KB).

The segment summary metadata contains information about each segment such as the segment checksum, sequence number, and the physical to logical mapping (P2L) for GC. It is located in the last block of a segment taking up 4 KB per segment. The metadata overhead for segment summary depends on SWAN's configuration. For example, for a 1 MB segment size, which is the size used in our experiments, segment summary takes up 0.39% of the storage space (i.e., 4 KB per 1 MB).

In the SWAN and Log-RAID prototypes, we maintain the entire metadata structures in DRAM, assuming that their contents are backed up by built-in batteries in the server. Owing to their huge size, however, keeping all of the data structures in DRAM could be burdensome, in terms of cost and energy. To address this, on-demand mapping that only keeps popular mapping entries in DRAM while storing the rest in SSDs can be considered. However, we do not consider this in this study.

4.2 Optimizing GC using TRIM

TRIM is used to further optimize GC. Once valid pages in a victim segment are written back to the new segment, then the victim segment is TRIMmed. This is efficient as the writing of the segments occur in a sequential manner and also, as the TRIM unit is large. With large segments being TRIMmed, the SSD firmware will perform erasures in an efficient manner. Thus, it helps SWAN achieve high performance regardless of SSD manufacturer.

5 Evaluation

In this section, we first present the evaluation results of micro-benchmarks to see how our design choices affect the behavior of SWAN and help to avoid GC interference. We then present the evaluation results of real-world workloads and compare the performance of SWAN to the traditional RAID0, RAID4, RAID5, and Log-structured management schemes (Log-RAID0 and 4) for an array of SSDs.

We evaluate SWAN on a Dell R730 server equipped with two Xeon E5-2609 CPUs and 64GB DRAM. We use 120GB Samsung 850 PRO SSDs of which measured peak read and write throughput is roughly 500MB/s and 400MB/s, respectively. The number of SSDs used differ from experiment to experiment as we describe later. We measure performance at the host system. Before any experiments for a particular

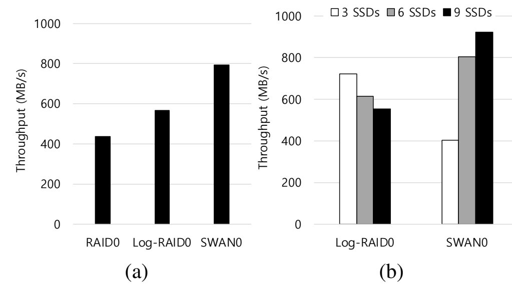


Figure 4: (a) Performance comparison of RAID0, Log-RAID0 and SWAN0 with 8 SSDs with SWAN0 configured as 4R-2SSD and (b) performance trend for Log-RAID0 and SWAN0 with 3, 6, 9 SSDs with SWAN configured as 3 R-groups with 1, 2, and 3 SSDs.

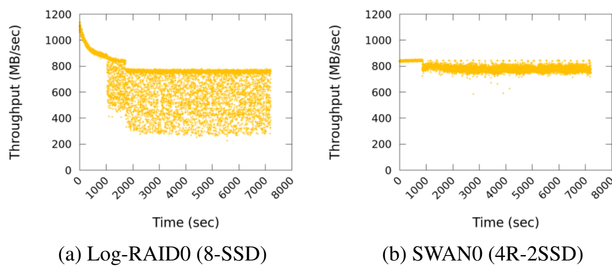
configuration, we go through a systematic cleaning and aging process; each SSD is first cleaned through formatting and TRIMming, and then the SSD is aged by making random writes to roughly 60% of the storage capacity.

5.1 Micro-benchmarks

We compare the performance of SWAN with two other AFA schemes, RAID and Log-RAID, all with RAID0 configurations. We denote each of these configurations as SWAN0, RAID0, and Log-RAID0, and the convention of attaching the suffix number representing the RAID type to the configuration will be used throughout hereafter. To understand their behavior especially under heavy GC, we make use of the FIO [6] benchmark issuing 8 KB random write (only) requests. To observe the raw performance, we disable any caching layers and directly issue writes to each scheme. Each experiment is conducted for two hours and its total footprint is roughly 12 TBs. Each experiment is performed more than 3 times and all results are within 6% of each other.

Figure 4(a) shows the results with 8 SSDs and SWAN configured as 4 R-groups of 2 SSDs each, which we denote as 4R-2SSD. Hereafter, this numbering convention will be used to represent SWAN configurations. The results show that RAID0 shows worst performance because it generates random writes and incurs high GC overhead inside the SSD. While Log-RAID0 transforms random writes to bulk, sequential writes, its performance is slower than SWAN0. The reason is Log-RAID0 requires GC to reuse log space, which issues read and write operations to all SSDs as illustrated in Figure 2. These GC related operations significantly degrade normal I/O operations. In contrast, SWAN0 shows close to full SSD throughput.

We then compare the performance of Log-RAID0 and SWAN0 with varying number of SSDs to understand how our partial aggregation of SSDs affects performance. We make use of 3, 6, 9 SSDs to configure Log-RAID0 and SWAN0, which, in turn, is configured as 3R-1SSD, 3R-2SSDs, and 3R-3SSDs using 3, 6, and 9 SSDs, respectively. As Figure 4(b) shows, surprisingly, the throughput of Log-



(a) Log-RAID0 (8-SSD) (b) SWAN0 (4R-2SSD)

Figure 5: Throughput of Log-RAID0 and SWAN0 over time

RAID0 degrades as more SSDs are used, while performance of SWAN0 improves. The reason behind this can be explained by the results shown in Figure 5, which shows (a) 8 SSDs configured as Log-RAID0 and (b) SWAN0 as 4R-2SSD. The results in this figure show that even with only 2 SSDs for SWAN, performance is actually better than Log-RAID0 with 8 SSDs. Furthermore, it shows that the throughput of Log-RAID0 fluctuates significantly, while SWAN0 shows high, sustained write performance that is proportional to the 2 SSDs in the R-group. The reason for throughput degradation with Log-RAID is that normal I/O and GC I/O interfere with each other. When normal I/O and GC I/O requests are being served by the same SSD, the latency of each I/O operation increases. As more SSDs get involved, the throughput of Log-RAID0 degrades because GC performance is limited by the slowest SSD [18].

5.2 Analysis of GC Behavior

We further analyze the performance degradation caused by AFA-level GC. For the random write workload of Figure 5, we plot the read/write throughput of each SSD for Log-RAID0 and SWAN0 in Figure 6. Note that all reads here are those issued for GC, and thus, we can observe the negative effect on overall performance due to such read operations.

Figure 6(a) shows the results for Log-RAID0. We see that all SSDs are involved in write operations throughout its execution. GC operations of Log-RAID begins from ⓐ in the figure, and performance starts to fluctuate from that point. This is also the point where read maintains a steady bandwidth overhead. (Though there is a line for read before this point, the value is 0.) The total amount of write requests up to this point closely coincides with the total RAID capacity. Once disk space becomes exhausted, GCs are triggered, and we observe performance deterioration and fluctuation. This observation lasts until the end of our experiment.

Figure 6(b) shows the performance for SWAN0. Here, in contrast to those of Figure 6(a), we see that the throughput of SSDs 1 and 2 that comprise the front-end R-group is close to 400MB/s, the maximum throughput of the SSD, as they are the ones receiving the write requests. The performance offered to the user is the aggregate of the two SSDs, which is roughly 800MB/s. Once the free segments in the first two SSDs are exhausted, SSD 3 and 4 become the front-end R-group and the old front-end R-group becomes the back-end. When SSD 7 and 8 become the front-end R-group, SWAN starts performing GC by selecting the SSDs with victim segments based on the greedy policy. In the figure, the R-group denoted by ⓑ is the front-end and the one denoted ⓒ is the selected back-end R-group that is performing GC. Note from ⓒ (and the magnified circle) that only SSDs performing garbage collection is incurring reads. All other SSDs neither incur reads or writes (except the ones of the front-end R-group). These front-end and back-end transitions are repeated throughout the experiments.

To quantitatively understand how SWAN GC behaves in runtime, we analyze the utilization of victim segments (i.e., the ratio of valid pages in a victim segment) and the number of free segments in the SWAN array for 80 minutes. Recall that as our workload continuously writes to the front-end R-group and GC on the back-end must continuously be performed in the background to maintain stable performance. From Figure 6(b), we observe that starting from around 800 seconds, GC starts to occur. Figure 7 shows that initially the utilization of the selected victim segments are 0, but then start to increase. The results show that, eventually, the utilization of the victim segment and the number of free segments are converging. This is because data is being overwritten and thus, the back-end R-group is likely to have many obsolete data. Such convergence shows that free segment generation through GC in SWAN is stable and does not interfere with the writes occurring in the front-end R-group.

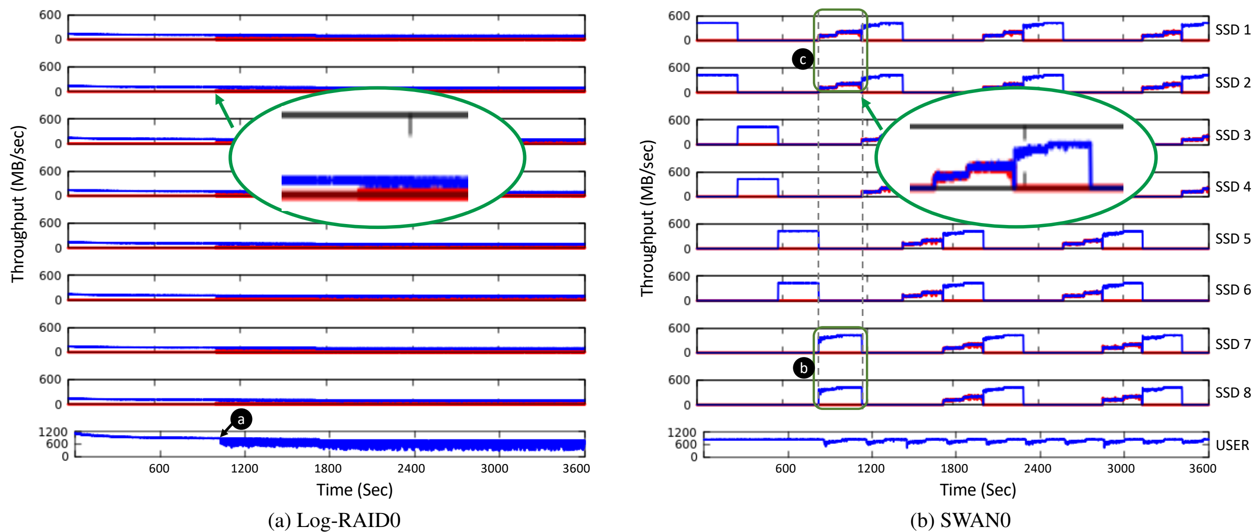
5.3 Real-world Workload

To see how effective SWAN is in a real-world setting, we experiment with the YCSB benchmark [11] on RocksDB [5]. For these experiments, we use RAID4 and 5 configurations, which is different from previous sections, to test SWAN in a more realistic setting. In particular, we use 9 SSDs with RAID4, RAID5, Log-RAID4, and SWAN4, which is configured as 3R-3SSD with 2 data SSDs and 1 parity SSD per R-group.

Table 3: I/O characteristics of YCSB benchmark

YCSB	Load	Run			
		A	B	C	D
Read	-	32GB	60GB	64GB	60GB
Update	-	32GB	3GB	-	-
Insert	64GB	-	-	-	3GB
R:W ratio	0:100	50:50	95:5	100:0	95:5

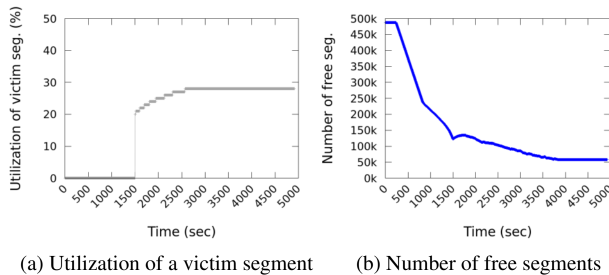
The workload characteristics of YCSB is summarized in Table 3, which includes the amount of data accessed by three different operations, Read, Update, and Insert, as well as the read/write ratios. Note that all the YCSB benchmarks consist of two phases: load and run phases. The load phase is responsible for creating the entire data set in the database, thus involves a large number of Insert operations. The run phase



(a) Log-RAID0

(b) SWAN0

Figure 6: Throughput of Log-RAID0 and SWAN0 for random write workload used in Figure 5. Top eight rows are the write throughput for each SSD and they include not only user requests but also GC incurred by each scheme. The bottom row shows the aggregate throughput of each scheme. The blue (upper) line denotes write throughput and the red (lower) line denotes the read incurred by GC. The SWAN configuration here is the same as that of Figure 5.



(a) Utilization of a victim segment (b) Number of free segments

Figure 7: Utilization of a victim segment and the number of free segments for SWAN0 with 8KB size random write workload for 80 minutes

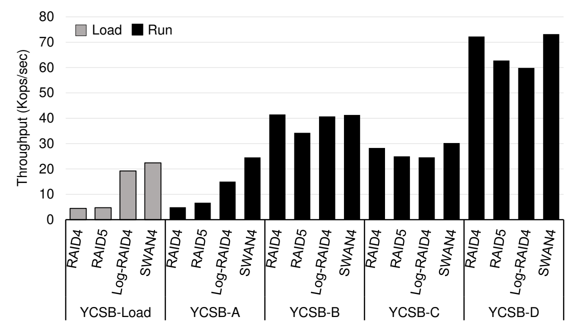


Figure 8: Throughput comparison for RAID4, RAID5, Log-RAID4, and SWAN4 for YCSB benchmark.

executes a specific workload (YCSB-A through YCSB-D) with different I/O patterns on the created data set.

Overall Performance: Figure 8 shows the overall throughput results. The results show that SWAN4 outperforms RAID4, RAID5, and Log-RAID for all the workloads. In the Load phase where almost all of the requests are writes, SWAN exhibits over 4× higher throughput compared to RAID-4/5 and even performs 17% better than Log-RAID. This is due to the fact that SWAN4 maintains sufficient free space to serve incoming writes immediately without interference by GC. In particular, for the YCSB-A workload where the workload is composed of reads and updates, SWAN4 performs significantly better than the other schemes, including Log-RAID4. Even in the other workloads, which are read dominant, SWAN4 performs slightly better or similar compared to the other schemes. As read is more latency-sensitive, we now further analyze read latency.

Read Latency: We now consider the effect of SWAN on read latency. As shown in Figure 9, the average read latency of SWAN is similar to or better than the other schemes. Moreover, as illustrated in Figure 10, SWAN exhibits much shorter tail latency compared to others across all of the YCSB benchmarks. This is because, in RAID4/5 and Log-RAID4, read requests are often blocked by AFA-level or SSD-level GC. In particular, we find that even under read-dominant workloads (YCSB-C and YCSB-D), SWAN4 exhibits shorter read latency. The reason for this is due to background GC. More specifically, recall that in all our experiments we include a systematic cleaning and aging process for the array of SSDs. We find that AFA-level GC (for Log-RAID4 and SWAN4) continues for a considerable length of time (roughly 15 minutes), while SSD-level GC [55] contin-

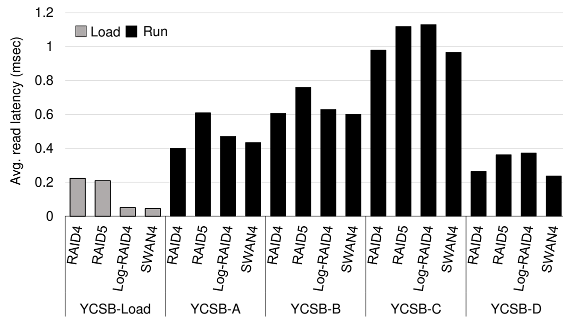


Figure 9: Average read latency comparison for RAID4, RAID5, Log-RAID4 and SWAN4 for YCSB benchmark

Table 4: Read requests in SWAN for YCSB-A workload

	# of requests (million)	Avg. latency (usec)
Front-end	20.3	98
Back-end (Idle)	25.9	88
Back-end (GC)	3.93	103

ues even further, which interfere with read requests. Fortunately, SWAN spatially separates GC so that it occurs only in one R-group, which enables us to effectively hide interference by GC. This argument is supported by Figure 11, which depicts the latency distribution of read requests for YCSB-C. Unlike the other schemes where we observe high latency spikes, SWAN shows fairly stable and consistent read latency.

To further understand SWAN's impact on read latency in more detail, we measure the latency of reads served by three different types of R-groups in SWAN: the frond-end, the idle back-end (that does not perform GC), and the busy back-end (that is performing GC), as shown in Table 4. As expected, the idle back-end provides the shortest read latency. The frond-end, on the other hand, is responsible for handling user writes, and thus it provides longer read latency than the idle back-end. Finally, the read latency on the busy back-end shows worst performance as it is more likely to be delayed by the erase and the write operations incurred by GC.

Table 4 also shows the number of reads handled by the three R-groups with the YCSB-A workload. We observe that 92% of the read requests are serviced by the idle back-end (52%) and the front-end (40%). Only 8% of the reads are destined for the busy back-end group that is performing GC. This skewed data access is due to YCSB's I/O pattern model that is based on the Zipf distribution, which is typically observed in many data-center applications [8, 14]. We find that, under the Zipf distribution with temporal locality, the front-end and the idle back-end R-groups are likely to receive more reads because they hold recently written data as we delay GC of R-groups as much as possible (as depicted in Figure 6). The frond-end receives a relatively smaller number of read requests than the idle back-end because many read requests are hit and served by the OS page cache holding data that were recently written but have not yet been evicted to the front-end R-group. The busy back-end contains old data, so only few read requests are directed to that R-group.

5.4 Analysis with an open-channel SSD

In AFA systems with RAID, I/O requests can be unexpectedly delayed if SSD-level GC is triggered. In particular, GC-blocked read I/Os are considered to be the root cause of long tail latency [60]. Unlike existing AFA systems, SWAN suffers less from SSD-level GC because it writes all the data in an append-only manner, thereby avoiding valid page copies for GC inside an SSD.

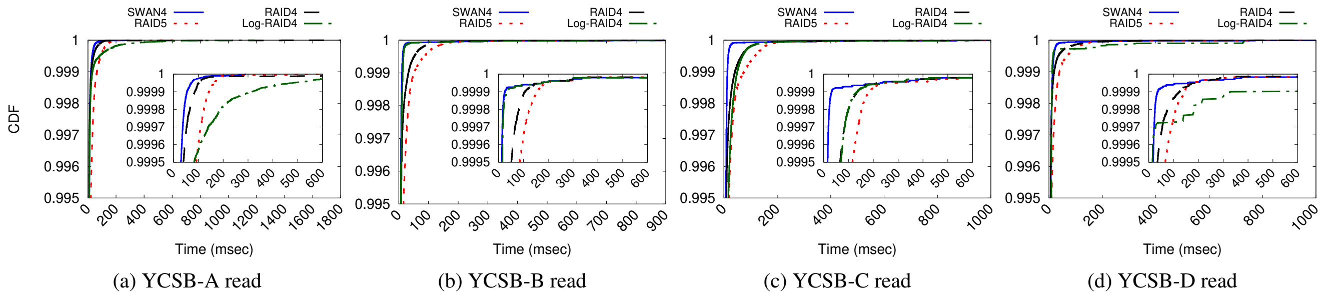
In this section, we quantitatively analyze the benefits of SWAN on individual SSDs, in terms of tail latency. Since we cannot modify and analyze the internals of off-the-shelf SSDs, we implement a custom page-level FTL scheme on an open-channel SSD [34]. From two different settings, SWAN0 and RAID0 with six SSDs, we collect block I/O traces of FIO random read/write workloads, and then replay the traces atop our open-channel SSD. We integrate a performance profiler with the custom FTL and monitor and collect detailed FTL activity statistics including page reads/writes, block erasures, as well as elapsed times for serving host reads and writes.

Figure 12 depicts the latency CDF measured in the open-channel SSD. The read and write latencies of NAND chips in the open-channel SSD is around 100*u*s and 500*u*s, respectively. SWAN0 shows shorter latency and shorter tail compared to RAID0 throughout its execution. This indicates that I/O performance of RAID0 is deteriorated by the extra page copies for internal GC. Consequently, the results confirms that SWAN is effective in reducing SSD-level GC overhead.

6 Discussion

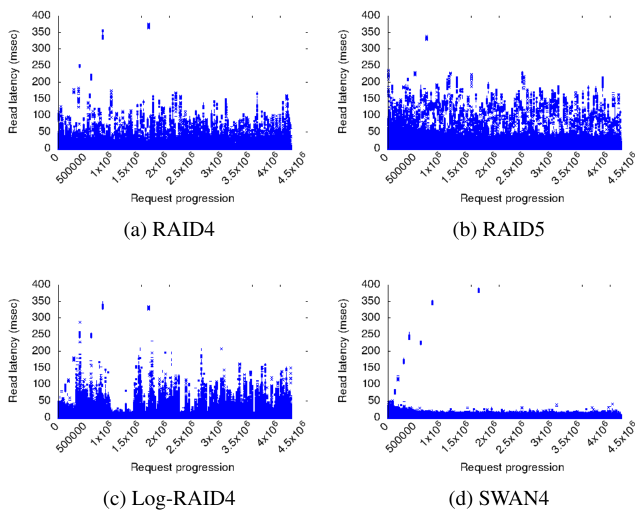
Benefits with simpler SSDs: The main design principle of SWAN is minimizing the performance interference caused by SSD-level GC and AFA-level GC. We think this opens opportunities to save cost and power consumption without compromising performance by adopting SSDs with simpler design. We expect that the main benefits of the simpler SSD will come from 1) smaller DRAM size, 2) FTL implemented on a low-power ARM core or hardware, and 3) smaller over-provisioning space (OPS).

A high-end modern SSD today requires an FTL (SSD firmware), a large amount of DRAM (e.g., 0.5-16 GB for mapping tables [45, 49]), high-end processors to run its space management and garbage collection algorithm (e.g., multi-core ARM processor [19]) along with additional over-provisioning space (OPS) (e.g., extra 6.7% to 28% of flash capacity just for OPS [46, 47]) to reduce garbage collection overhead. However, SWAN does not rely on such a sophisticated, powerful FTL. SWAN sequentially writes data to segments and TRIMs a large chunk of data in the same segment at once. This implies that an SSD receives sequential write



(a) YCSB-A read (b) YCSB-B read (c) YCSB-C read (d) YCSB-D read

Figure 10: CDF of read latency for YCSB benchmark. The tail latency of SWAN is shortest in all workloads. In particular, at 99.9th or higher latency, SWAN shows much shorter latency than others.

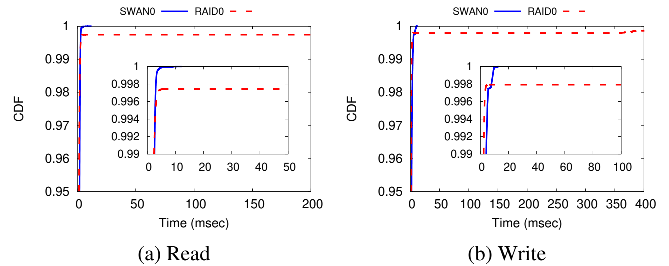


(a) RAID4 (b) RAID5

(c) Log-RAID4 (d) SWAN4

Figure 11: Read latency distribution of YCSB-C

streams all the time from the host, which will be obsolete together later. Under such workloads, it is only necessary for an SSD to carry out block erasures to reclaim fully invalidated flash blocks, and thus complicated media management algorithms like address remapping and garbage collection are not needed. On the SSD side, actually, a simple block-level FTL is sufficient to support SWAN's workloads. By making the design of FTL simpler, we can reduce cost for DRAM and the processor inside the SSD and save power consumption as well. For example, a page-level FTL scheme requires roughly 1 GB of memory for a 1 TB SSD to manage the mapping information [45]. However, in our experience of implementing the block-level FTL for SWAN, only 8 MB of DRAM is required for address mapping. Also, for SSDs deployed with SWAN, they do not require a powerful processor to run sophisticated FTL algorithms such as hot-cold separation and multi-streamed I/O management [27] that are designed to reduce GC overhead. We expect a single low-power ARM core or even hardware logic to be enough to manage NAND flash with SWAN. Finally, SSDs used in SWAN do not need to reserve large OPS, which is critical to reduce



(a) Read (b) Write

Figure 12: Latency CDF of FIO random read/write workloads measured in open-channel SSD for RAID0 and SWAN0

GC overhead. This has the benefit of improving the effective storage capacity provided to users.

Effect of NVRAM: As a remedy for GC overhead, one might argue that NVRAM could be used as a write buffer to accommodate incoming writes while the underlying SSDs are busy doing GC. Using NVRAM, however, is costly because it requires expensive battery-backed DRAM. Thus, even high-end AFA controllers have only few GBs of NVRAM (e.g., 8-64 GB [3]) and use it to improve data persistence and consistency, for example, by keeping user data for a few seconds before a new consistency point starts [56]. However, considering various factors such as cost of NVRAM, the high bandwidth of the AFA network, the ever-increasing working-set size, as well as the limited DIMM slots, we think using NVRAM to buffer large amounts of user data for an extended period of time to maintain high throughput and hide SSD-level GC is not an easily acceptable solution.

7 Related Work

Reducing GC overhead: Considerable effort have been made at various layers in the storage stack to reduce GC overhead in flash storage, including at the file system [32, 39, 41], the I/O scheduler [24, 29], buffer replacement [26], and the SSD itself [17, 27, 28]. These efforts, likewise, alleviate GC overhead through reduced write amplification, but

do not completely remove or hide GC overhead.

The recently proposed TTFlash almost eliminates GC overhead by exploiting a combination of current SSD technologies including a powerful flash controller, the Redundant Array of Independent NAND (RAIN) scheme, and the large RAM buffer in SSD internals [60]. By making use of the timely technologies in SSD, Tiny-tail handles I/O requests with almost no-GC scenario, with the caveat that the copy-back operation must be supported. While Tiny-tail is an SSD internal approach, it is different from what we propose as, first, we target an array of SSDs and second, we can make use of any commodity SSD, though a SWAN optimized, simpler SSDs would be most efficient.

GC preemption: Preemption is another way to decouple GC impact from user requests. GC preemption is a means of virtually postponing GC to avoid conflicts between GC and user requests. A number of studies, including an industry standard, have been conducted in this direction [33, 57, 59]. However, GC preemption is prone to failure for various reasons such as excessive write requests or ill-chosen GC policies [60].

Array of Flash/SSDs: There have been studies to address GC impact in arrays of SSDs. Flash on Rails [51] and Harmonia [30] are SSD-based array schemes suggested to resolve the GC problem. Flash on Rails separates read and write requests on different physical disks to separate the read request handling SSD from the GC handling SSD. This is a similar approach as our work, with the difference being that we consider a large scale, network connected storage system while Flash on Rails maintain at least one replica SSD for servicing read requests. It basically differs from SWAN in physical data placement and redundancy level. In large capacity storage devices such as an AFA system, this doubling of space is subject to deployment constraints. In contrast, in Harmonia, the host OS synchronizes the GC of all SSDs to prevent request blocking from unaligned GC for an array of SSDs. This approach does not remove or hide GC, but synchronizes GC to reduce its negative effect.

Gecko: The work most similar to ours is Gecko, which was designed for an array of HDDs [50]. Gecko is similar to SWAN in that it views the chain of HDDs as a log with new writes being made to the tail of the log to reduce disk contention by GC. SWAN advances this idea especially in the context of AFA, which is SSD-based. The key differences between Gecko and SWAN in terms of storage media can be summarized are as follows. 1) SWAN provides a guide to organizing of an array of SSDs based on the analytical model that reflects the characteristics of commercial SSD devices. 2) SWAN introduce the most efficient way to use SSDs in AFA through writing large amount of data sequentially and trimming, which is an SSD-only feature. 3) GC preemption is employed for serving read requests. 4) SWAN provides implications for a cost effective SSD design for AFA.

In terms of system organization, unlike Gecko, which uses a one-dimensional array of HDDs, SWAN manages SSDs in two dimensions to spatially separate GC writes from first-class writes and to achieve higher aggregated storage throughput than the network throughput. Also, Gecko has to prevent interference by read operations because it targets HDDs, where a read operation can also move the disk head.

Exposing flash to host: LightNVM [7] and Application-managed Flash [34] attempt to eliminate GC overhead by letting the host software manage the exposed flash channel. These approaches are similar to our method in that GC is being managed by the host, but they are different in that they do not decouple the I/Os for GC and those requested by user applications. Hence, even though these approaches reduce GC impact by directly controlling the flash devices from the host, GC is required in managing the flash device. SWAN, on the other hand, hides GC overhead through host controlled GC in an array of SSDs.

8 Conclusion

We presented a novel all flash array management scheme, named SWAN (Spatial separation Within an Array of SSDs on a Network). Our work was motivated by key observations that aggregating a number of SSDs is sufficient to surpass the network bandwidth. However, burdensome garbage collection together with all flash array software prevented us from realizing optimal performance by making it difficult to fully saturate the peak network bandwidth. In an attempt to overcome this problem, SWAN decoupled GC I/Os from normal ones by partitioning the SSD array into two mutually exclusive groups and by using them for different purposes in a serial manner: 1) serving incoming writes or 2) performing GC in the background. This spatial separation of SSDs enabled us to hide costly GC overheads, providing GC free performance to the applications. Moreover, using an analytical model, we confirmed that SWAN guaranteed no GC interference I/Os at all times if two mutually exclusive groups were properly partitioned. Our evaluation results showed that SWAN offered consistent I/O throughput at close to the maximum network bandwidth and that read latency also improved.

Acknowledgment

We would like to thank our shepherd Patrick P. C. Lee and the anonymous reviewers for their constructive comments. This work was supported by in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2019R1A2C2009476), by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1E1A1A01077410). Sam H. Noh is the corresponding author.

References

[1] EMC XtremIO X2 Specification. https://www.dellemc.com/resources/en-us/asset/data-sheets/products/storage-2/h16094-xtremio-x2-specification-sheet-ss.pdf.

[2] HPE 3PAR StoreServ Specification. https://h20195.www2.hpe.com/V2/GetDocument.aspx?docname=4AA3-2542ENW.

[3] NetApp All Flash FAS. https://goo.gl/1D9dmT.

[4] NetApp SolidFire Specification. https://www.netapp.com/us/media/ds-3773.pdf.

[5] RocksDB: A persistent key-value store. https://rocksdb.org/.

[6] AXBOE, J. FIO: Flexible I/O Tester. https://github.com/axboe/fio.

[7] BJÃŽRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2017), pp. 339–353.

[8] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment 5*, 12 (Aug. 2012), 1802–1813.

[9] CHIUEH, T.-C., TSAO, W., SUN, H.-C., CHIEN, T.-F., CHANG, A.-N., AND CHEN, C.-D. Software Orchestrated Flash Array. In *Proceedings of International Conference on Systems and Storage (SYSTOR)* (2014), pp. 14:1–14:11.

[10] COLGROVE, J., DAVIS, J. D., HAYES, J., MILLER, E. L., SANDVIG, C., SEARS, R., TAMCHES, A., VACHHARAJANI, N., AND WANG, F. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2015), pp. 1683–1694.

[11] 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 (SoCC)* (2010), pp. 143–154.

[12] DAVIS, R. The Network is the New Storage Bottleneck. https://www.datanami.com/2016/11/10/network-new-storage-bottleneck/, 2016.

[13] DESNOYERS, P. Analytic Models of SSD Write Performance. *ACM Transactions on Storage 10*, 2 (Mar. 2014), 8:1–8:25.

[14] DI, S., KONDO, D., AND CAPPELLO, F. Characterizing Cloud Applications on a Google Data Center. In *Proceedings of International Conference on Parallel Processing (ICPP)* (2013), pp. 468–473.

[15] EDSALL, T., KASER, R., MEYER, D., SEQUEIRA, A., AND WARFIELD, A. Networking is Fast Becoming the Bottleneck for Storage and Compute, How Do We Fix It? https://www.onug.net/town-hall-meeting-\networking-is-fast-becoming-the-bottleneck-for-\storage-and-compute-how-do-we-fix-it/, Open Network User Group, 2016.

[16] GAL, E., AND TOLEDO, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Survey 37*, 2 (2005), 138–163.

[17] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 229–240.

[18] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 263–276.

[19] HITACHI. Hitachi Accelerated Flash 2.0. https://www.hitachivantara.com/en-us/pdf/white-paper/hitachi-white-paper-accelerated-flash-storage.pdf.

[20] IDC. The Digital Universe of Opportunities: Rich Data and the Increasing Value of The Internet of Things. https://www.emc.com/leadership/digital-universe/2014iview/index.htm, 2014.

[21] IOANNOU, N., KOURTIS, K., AND KOLTSIDAS, I. Elevating commodity storage with the SALSA host translation layer. In *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018), pp. 277–292.

[22] JIN, Y. T., AHN, S., AND LEE, S. Performance Analysis of NVMe SSD-Based All-flash Array Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), pp. 12–21.

[23] JUNG, M., CHOI, W., SHALF, J., AND KANDEMIR, M. T. Triple-A: A Non-SSD Based Autonomic All-flash Array for High Performance Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 441–454.

[24] JUNG, M., CHOI, W., SRIKANTAIAH, S., YOO, J., AND KANDEMIR, M. T. HIOS: A Host Interface I/O Scheduler for Solid State Disks. In *Proceedings of the Annual International Symposium on Computer Architecuture (ISCA)* (2014), pp. 289–300.

[25] KAISLER, S., ARMOUR, F., ESPINOSA, J. A., AND MONEY, W. Big Data: Issues and Challenges Moving Forward. In *Proceedings of the 46th Hawaii International Conference on System Sciences (ICSS)* (2013), pp. 995–1004.

[26] KANG, D. H., MIN, C., AND EOM, Y. I. An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices. In *Proceeding of the 22nd IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2014), pp. 239–248.

[27] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The Multi-streamed Solid-State Drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[28] KIM, H., AND AHN, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 16:1–16:14.

[29] KIM, J., OH, Y., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Disk Schedulers for Solid State Drives. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT)* (2009), pp. 295–304.

[30] KIM, Y., ORAL, S., SHIPMAN, G. M., LEE, J., DILLOW, D. A., AND WANG, F. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2011), pp. 1–12.

[31] KWON, H., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Janus-FTL: Finding the Optimal Point on the Spectrum between Page and Block Mapping Schemes. In *Proceedings of the International Conference on Embedded Software (EMSOFT)* (2010), pp. 169–178.

[32] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[33] LEE, J., KIM, Y., SHIPMAN, G. M., ORAL, S., AND KIM, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32*, 2 (2013), 247–260.

[34] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND. Application-Managed Flash. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 339–353.

[35] MAO, B., JIANG, H., WU, S., TIAN, L., FENG, D., CHEN, J., AND ZENG, L. HPDA: A Hybrid Parity-based Disk Array for Enhanced Performance and Reliability. *ACM Transactions on Storage 8*, 1 (Feb. 2012), 4:1–4:20.

[36] MARJANI, M., NASARUDDIN, F., GANI, A., KARIM, A., HASHEM, I. A. T., SIDDIQA, A., AND YAQOOB, I. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access 5* (2017), 5247–5261.

[37] MARRIPUDI, G., AND LLKER CEBELI. How Networking Affects Flash Storage Systems. Flash memory summit 2016.

[38] MENON, J. A Performance Comparison of RAID-5 and Log-structured Arrays. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)* (1995), pp. 167–178.

[39] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2012), pp. 139–154.

[40] NANAVATI, M., SCHWARZKOPF, M., WIRES, J., AND WARFIELD, A. Non-volatile storage. *ACM Queue 13*, 9 (2015), 20:33–20:56.

[41] OH, Y., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the Annual Haifa Experimental Systems Conference (SYSTOR)* (2010), pp. 2:1–2:9.

[42] OH, Y., LEE, E., HYUN, C., CHOI, J., LEE, D., AND NOH, S. H. Enabling Cost-Effective Flash Based Caching with an Array of Commodity SSDs. In *Proceedings of the Annual Middleware Conference (Middleware)* (2015), pp. 63–74.

[43] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[44] REINSEL, D., GANTZ, J., AND RYDNING, J. Data Age 2025: The Evolution of Data to Life-Critical. https://www.seagate.com/our-story/data-age-2025/, 2017.

[45] SAMSUNG. 960PRO SSD Specification. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/.

[46] SAMSUNG. Over-provisioning: Maximize the Lifetime and Performance of Your SSD with Small Effect to Earn More. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_845DC_04_Over-provisioning.pdf.

[47] SAMSUNG. Samsung NVMe SSD. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/SAMSUNG_Memory_NVMe_Brochure_web.pdf.

[48] SAMSUNG. SSD 970 EVO NVMe M.2 1TB. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-1tb-mz-v7e1t0bw/.

[49] SAMSUNG. PM1633a NVMe SSD. https://goo.gl/PkRpKf, 2016.

[50] SHIN, J.-Y., BALAKRISHNAN, M., MARIAN, T., AND WEATHERSPOON, H. Gecko: Contention-oblivious Disk Arrays for Cloud Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 285–298.

[51] SKOURTIS, D., ACHLIOPTAS, D., WATKINS, N., MALTZAHN, C., AND BRANDT, S. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2014), pp. 463–474.

[52] THE ECONOMIST. Data is giving rise to a new economy. https://www.economist.com/news/briefing/21721634-how-it-shaping-up-data-giving-rise-new-economy, 2017.

[53] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 145–158.

[54] WIKIPEDIA. mdadm. https://en.wikipedia.org/wiki/Mdadm.

[55] WIKIPEDIA. Write amplification. https://en.wikipedia.org/wiki/Write_amplification.

[56] WOODS, M. Optimizing Storage Performance and Cost with Intelligent Caching (NetApp's White Paper). https://logismarketpt.cdnwm.com/ip/elred-netapp-virtual-storage-tier-optimizing-storage-performance-and-cost-with-intelligent-caching-929870.pdf, 2010.

[57] WU, G., AND HE, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2012), pp. 10–10.

[58] WU, S., ZHU, W., LIU, G., JIANG, H., AND MAO, B. GC-Aware Request Steering with Improved Performance and Reliability for SSD-Based RAIDs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), pp. 296–305.

[59] WU, W., TRAISTER, S., HUANG, J., HUTCHISON, N., AND SPROUSE, S. Pre-emptive Garbage Collection of Memory Blocks, Jan. 7 2014. US Patent 8,626,986.

[60] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-Tail Flash: Near-Performance Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2017), pp. 15–28.

Practical Erase Suspension for Modern Low-latency SSDs

Shine Kim[†‡], Jonghyun Bae[†], Hakbeom Jang*, Wenjing Jin[†], Jeonghun Gong[†]
Seungyeon Lee[‡], Tae Jun Ham[†], Jae W. Lee[†]
[†]*Seoul National University,* **Sungkyunkwan University,* [‡]*Samsung Electronics*

Abstract

As NAND flash technology continues to scale, flash-based SSDs have become key components in data-center servers. One of the main design goals for data-center SSDs is low read tail latency, which is crucial for interactive online services as a single query can generate thousands of disk accesses. Towards this goal, many prior works have focused on minimizing the effect of garbage collection on read tail latency. Such advances have made the other, less explored source of long read tails, *block erase operation*, more important. Prior work on erase suspension addresses this problem by allowing a read operation to interrupt an ongoing erase operation, to minimize its effect on read latency. Unfortunately, the erase suspension technique attempts to suspend/resume an erase pulse at an arbitrary point, which incurs additional hardware cost for NAND peripherals and reduces the lifetime of the device. Furthermore, we demonstrate this technique suffers a write starvation problem, using a real, production-grade SSD. To overcome these limitations, we propose alternative *practical* erase suspension mechanisms, leveraging the iterative erase mechanism used in modern SSDs, to suspend/resume erase operation at well-aligned safe points. The resulting design achieves a sub-200μs 99.999th percentile read tail latency for 4KB random I/O workload at queue depth 16 (70% reads and 30% writes). Furthermore, it reduces the read tail latency by about 5× over the baseline for the two data-center workloads that we evaluated with.

1 Introduction

NAND flash-based SSDs offer superior throughput and average latency compared to those of hard disks and thus have become the de-facto standard for storage devices. However, SSDs have much greater performance variability than that of hard disks [11]. While SSDs can achieve very low average read latency (e.g., under 15μs [5]), their tail latency (e.g., 99.999th percentile) can be very long (e.g., over 10ms). Figure 1 demonstrates this issue with a real low-latency SSD.

One can mistakenly think that long tail read latency is a rare event that affects very few requests. However, such tail latency can play a considerable role in data-center computing because a single query (e.g., web search) may require thousands of disk reads across the data-center [2,7]. In such a case, a single long-latency disk access can lead to an increase in the overall query response time. Also, the chance of a query experiencing

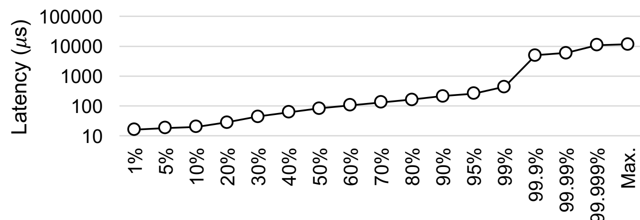


Figure 1: Read latency distribution of a PCIe 3×4 NVMe low-latency SSD [5] running 4KB random reads (70%) and writes (30%) workload with a queue depth of 16.

long disk tail latency is continuously increasing with the trend of ever-increasing data size.

To avoid such performance degradation in a data-center induced by SSD tail latency, minimizing the tail latency of a read operation is very important. Naturally, this requires tackling two significant sources of long read tails: garbage collection (GC) and erase operations. Recent prior works have already explored ways to minimize the effect of GC on read tail latency [6, 14, 37]. Furthermore, production-level low-latency SSDs — such as Samsung Z-SSD [38], which we base our work on — have already minimized the effect of GC on read operation [38] by allowing the user-initiated read to be processed between operations in a GC (i.e., read and program to copy valid pages from one block to another) [4, 20]. In contrast, much less attention has been paid to the effect of erase operation on read tail latency, primarily because erase latency is relatively small (e.g., 5ms) compared to the effect of GC on tail latency (e.g., can exceed 100ms without any optimization). However, with techniques minimizing this effect, erase latency is now becoming the most dominant component of read tail latency.

To control the effect of erase on tail latency, Wu et al. [35] proposed an erase suspension technique, which suspends an ongoing erase (and verify) pulse when a read request is issued to the same flash die. After processing the read request, the erase pulse resumes from the exact point at which it was suspended. However, the commodity NAND flash business is known to be extremely cost-sensitive, and this technique may increase the cost of NAND peripherals to generate an erase pulse of an arbitrary length and track the exact state of every erase. Furthermore, it can cause a serious NAND reliability problem and write starvation.

To address these limitations, we present *practical* erase suspension schemes for modern low-latency SSDs. Instead

of suspending/resuming an erase pulse at an arbitrary point, our work focuses on suspending/resuming the erase operation at well-aligned safe points by either i) aborting an ongoing erase operation immediately and resuming from the last safe point or ii) deferring the suspension of erase operation until the next safe point. Exploiting their trade-offs, we also introduce a timeout-based switching mechanism between the two mechanisms, to adapt to workload changes dynamically. This scheme enables modern low-latency SSDs to offer extremely low read tail latency on a wide range of workloads without causing any NAND reliability problem or write starvation.

Our contributions are summarized as follows:

- We are the first to identify the problems of NAND reliability and write starvation in the existing erase suspension scheme and demonstrate the latter on a real, production-grade SSD.
- We propose two practical erase suspension mechanisms, *immediate* erase suspension and *deferred* erase suspension. We also analyze the trade-offs between the two mechanisms and introduce a timeout-based switching policy between the two, to take the best of both as the workload changes.
- We demonstrate a significant reduction in read tail latency with the proposed erase suspension mechanisms on various workloads including Aerospike Certification Tool (ACT) [1] and TPC-C benchmark workloads [31].

2 Practical Erase Suspension

2.1 Motivation

To perform a NAND block erase, the incremental step pulse erasing scheme is a standard feature in modern SSDs [18]. Instead of utilizing a single, very high voltage pulse (e.g., 14V) for an erase, which has negative impact on NAND lifetime [12, 25, 29], this scheme performs an erase operation with several, discrete pulses (typically 5 or fewer), and each pulse has a higher nominal voltage than the previous one. Figure 2 illustrates this scheme. By verifying the set of erased cells between erase pulses and by applying higher voltage pulses to cells that are not erased yet, this scheme minimizes damage on NAND cells [12]. A single erase pulse consists of the following 3 stages: ① *voltage ramping* stage in which the erase pulse reaches the desired voltage, ② *erase execution* stage during which the voltage is stabilized and maintained, and ③ *voltage recovery* stage in which the erase voltage is reset for the erase-verify operation.

The erase suspension mechanism has been proposed to provide tight read tail latency by suspending an ongoing erase pulse at the arrival of a read request to be resumed later [35]. While effective in reducing read tail latency, this mechanism poses several implementation challenges in terms of NAND

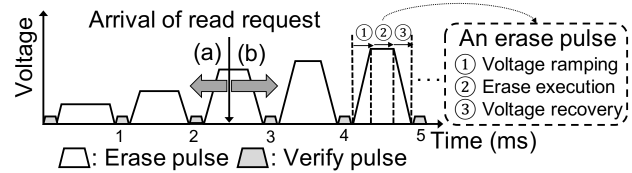


Figure 2: Incremental step pulse erase and practical erase suspension mechanisms: (a) Immediate erase suspension (I-ES) and (b) Deferred erase suspension (D-ES).

reliability and cost. First, an erase suspension adds to the cost of an erase an extra pair of voltage recovery (③) and ramping (①) stages for suspending and resuming the erase pulse. These additional stresses caused by ramping up and down the voltage degrade the endurance of NAND [12, 25, 29]. For example, a recent case study using sub-20nm TLC NAND shows that the *raw bit error rate (RBER)* is increased by 14.4% with only 1000 erase suspensions [26]. The increased RBER leads to an increase in *uncorrectable bit error rate (UBER)* even with error correction, to eventually reduce the lifetime of SSD [3].

Another limitation of the existing erase suspension scheme is that it requires the capability to suspend and resume an erase pulse at an arbitrary point. This increases the cost for NAND peripherals to generate a pulse of an arbitrary length, track the exact state of each suspended erase, and recover the peripheral state to resume. As NAND cells continue to scale with the introduction of 3D NAND, NAND peripherals are becoming a scalability bottleneck to a greater extent [19, 24, 27]. Considering the extreme cost sensitivity of the commodity NAND business, this is a significant drawback.

Instead, our erase suspension scheme allows erase suspension only at the beginning or the end of each erase step. When a read request arrives while an erase pulse is still asserted, our scheme asserts an erase suspension command that either i) aborts the ongoing erase pulse and forfeits the current erase step progress to serve the read request immediately (named *immediate erase suspension*) or ii) finishes the ongoing erase step and serves the read request (named *deferred erase suspension*). The next subsections discuss each option in detail and also present an adaptive switching scheme between our two proposed mechanisms.

2.2 Immediate Erase Suspension (I-ES)

One way to serve an incoming read request during the erase pulse is to immediately terminate the ongoing erase step and to forfeit the progress (Figure 2(a)). Then, after serving the read request(s), the erase operation can resume from the beginning point of the current erase step. We call this scheme *Immediate Erase Suspension (I-ES)* since it immediately cancels the ongoing erase step.

In effect, I-ES is a practical variant of the original erase suspension [35] with the following two changes. To improve NAND reliability, a verify pulse is applied before resuming a suspended erase pulse, exploiting the incremental step

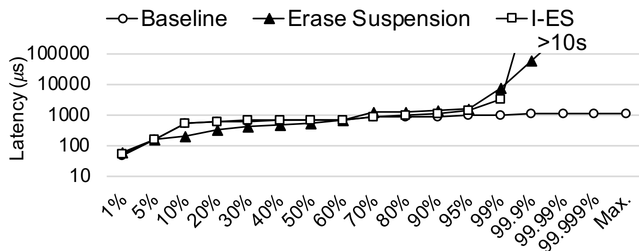


Figure 3: Write starvation experiment results. This workload consists of two threads where one thread continuously generates 128KB read requests, while another thread continuously generates 128KB write requests (QD 1 for both). Baseline and I-ES use real low-latency SSD [5], and Erase Suspension [35] uses an SSD simulator [33].

pulse erasing scheme in Section 2.1. As each NAND cell in a NAND block has different erase timing, some NAND cells get erased more quickly than others [12,25,26,29]. Therefore, applying the same pulse to all NAND cells when resuming the erase pulse results in over-erasure of some cells, which puts unnecessary stress on already erased cells and harms the reliability of NAND. A verify pulse, before resuming the erase pulse, detects already erased cells to not inflict unnecessary stress on these cells in a NAND block. Furthermore, to keep the cost of NAND peripherals low, a whole step pulse is asserted at every resumption, instead of the remaining step pulse time. To generate an erase pulse of an arbitrary length, a variable-length pulse generator with fine-grained control must be placed on a NAND device. However, today's commodity NAND does not provide such a mechanism. In contrast, I-ES does not require such changes keeping the cost of NAND low.
Advantages and Disadvantages. Since the erase step is canceled immediately, the read command can always be guaranteed the highest priority. The read command does not experience any other delay caused by an erase operation except for a fixed latency (e.g., $\sim$100μs) that tries to cancel the ongoing erase pulse (③ in Figure 2).

However, this scheme is problematic. With the continuous incoming read requests, an erase step may be repeatedly canceled, preventing write requests (which are dependent on the erase requests) from completion. To confirm this behavior, we modified the firmware of the real, production-grade low-latency SSD [5] to implement this scheme. Figure 3 shows the outcome of this experiment. As expected, a few write operations cannot finish for a long time because the preceding erase operation is continuously canceled by incoming read requests. As a result, its tail latency keeps increasing until the end of the workload.

The erase (and write) starvation problem occurs because the latency of erase suspension/resuming operation is longer than the incoming rate of read requests on the NAND chip. For example, if the throughput of PCIe Gen 3$\times$4 NVMe interface is 3.2GB/s (i.e., the incoming read rate is 1.19μs/4KB) and the erase suspension/resuming overhead is 100μs, it is possible for a read request to arrive while the erase suspension/resumption is happening. In such cases, the erase will immediately suspend again without making any forward progress, and thus the erase (and write) operation will starve. Note that the same problem manifests with the original erase suspension scheme [35], which differs from I-ES only for the duration of the asserted pulse at resumption, as shown in Figure 3. NVMe and PCIe specifications require the host to reset both hardware and software at write starvation (i.e., user's write requests timeout) [23, 28], which adversely affects system performance [22] and may eventually lead to a fatal system failure if repeated.

2.3 Deferred Erase Suspension (D-ES)

Another way to serve an incoming read request during the erase pulse is to let the read request wait until the current erase step finishes (Figure 2(b)). After that, in lieu of proceeding to the next erase step, the read request(s) is (are) served. Then, the erase operation resumes and continues to the next erase step. We call this *Deferred Erase Suspension (D-ES)* as it defers erase suspension until the end of the current erase step.
Advantages and Disadvantages. This mechanism lets the erase operation finish without incurring the write starvation problem. By guaranteeing a single erase step to be performed once the erase step is initiated, this mechanism guarantees forward progress for the erase operation.

Although this mechanism does not incur erase (and write) starvation, it can harm read tail latency by making read requests wait until the end of the current erase step (i.e., 1ms in our low-latency NAND). This scheme can also show worse read latency for bursty reads as D-ES adds extra latency to reads. However, such situations could be avoided by batching backlogged erases after serving the bursty reads first. This issue is addressed by T-ES, which switches adaptively between I-ES and D-ES.

2.4 Timeout-based Erase Suspension (T-ES)

Timeout-based Erase Suspension Policy. If it is possible to know the request pattern of an application *a priori*, one should use I-ES when the application is expected to have a phase with sparse read requests as in this case erase (and write) starvation does not occur as an erase is likely to make progress during the upcoming sparse read period. On the other hand, if an application is expected to have a steady stream of incoming read requests, the user should use D-ES as in this case, employing I-ES leads to an exponential increase in write tail latency (i.e., write starvation).

Unfortunately, in reality, it is not easy to predict future I/O access patterns without profiling runs. Thus, we propose a *Timeout-based Erase Suspension (T-ES)* scheme, which performs I-ES until the erase operation is delayed for Nms. If the erase operation is delayed for Nms (i.e., timeout happens), this scheme switches to D-ES mode to avoid potential erase (and write) starvation. Hopefully, this scheme finds a period to perform an erase operation without being interrupted by a

Table 1: Parameters used to model our low-latency SSD [5].

PCIe Gen 3×4 Lane, 240GB, NVMe SSD Device	
NAND Configurations	4 channels, 4 chips/channel, 1 die/chip
DRAM, Flash Speed Rate	1600MT/s, 1200MT/s (MT/s: Mega Transfers per Second [9])
FTL Schemes	Page Mapping, Preemptible GC [20]
Over-provisioning Ratio	7%
NAND Structure	
128Gb die capacity, 8 planes per die, 683 blocks per plane, 768 pages per block, 4KB page	
NAND Latency	
Read: 3μs, Program: 100μs, Block Erase: 1ms per step (5 steps), Erase Suspension Penalty: 100μs	

Table 2: Throughput and average latency of low-latency SSD prototype and MQSim running various I/O pattern workloads.

	Low-Latency SSD / MQSim	
Seq. read (256KB)	3300 / 3250 MiB/s	
Seq. write (256KB)	2700 / 3100 MiB/s	
	Kilo I/O per seconds	Average latency
Rnd. read (4KB, QD 1)	59 / 65 KIOPS	16.9 / 15.3μs
Rnd. write (4KB, QD 1)	61 / 66 KIOPS	16.4 / 15.2μs
Rnd. read (4KB, QD 32)	790 / 790 KIOPS	40.5 / 40.5μs
Rnd. write (4KB, QD 32)	61 / 66 KIOPS	524 / 484μs

read operation.

Choice of Erase Timeout Delay. T-ES covers a spectrum of policies between I-ES and D-ES, controlled by the value of N. If we set it to 0, this is equivalent to the base D-ES scheme. In contrast, if we set it to infinite, this is equivalent to I-ES scheme prone to erase (and write) starvation problem. In general, choosing a higher value offers more chance to provide better read tail latency by delaying an erase operation, but this choice leads to an increase in maximum write tail latency. On the other hand, choosing a smaller N makes it behave more like a D-ES, which provides smaller maximum write tail latency at the expense of an increased number of reads experiencing erase latency. T-ES provides a knob for the user to choose N based on her willingness to trade-off the maximum write tail latency for potential improvement in read tail latency. For example, if the user wants to achieve sub-100ms write tail latency and the maximum write delay occurring from a GC scheme (e.g., GC policy [6, 13–15, 17, 37], over-provisioning ratio [30]) is 35ms, the user should set N to be 64ms so that the total maximum write latency remains under 100ms. By default, we set N to 64ms for our experiments.

3 Evaluation

3.1 Methodology

Evaluation Framework. Although we utilized a real, prototype low-latency SSD [5] with modified firmware to perform some experiments (Figure 1 and Figure 3), it is not possible to utilize the real device to evaluate our presented schemes such as D-ES and T-ES since implementations of such policy require an extension to the interface between the SSD controller and NAND flash chips (e.g., new commands). For this

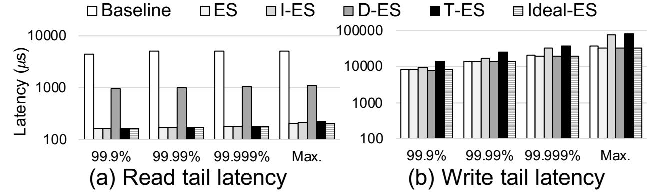


Figure 4: Read/write tail latency on 4KB random reads (70%) and writes (30%) workload.

reason, we use MQSim [33] for our experiments, which we extend so that it can accurately model low-latency NAND flash chips. In particular, we i) allow data cache manager and FTL to use strict 8 plane program so that it can achieve higher write throughput, ii) enable I/O scheduler to process the user read request as the highest priority, and iii) modify NAND flash controller and memory logic to model our practical erase suspension mechanisms. Table 1 summarizes the parameters used for our MQSim, while Table 2 compares the simulation results against those of a real low-latency SSD [5] for various I/O patterns as validation of the simulator. The average read/write latency and throughput from the simulator demonstrate only a 6% error on average (with 13% in the worst case) compared to measurements from the real device.

Evaluated Configurations. Throughout this section, we present evaluation results for the following configurations across different benchmarks (i.e., random 4KB access, ACT [1], and TPC-C [34]). Also, we use for all experiments the steady state pre-condition [32] in which the space of an SSD including the over-provisioning (OP) [30] area is full; this pre-condition helps evaluate the latency behaviors that can happen under the worst-case condition of read and write requests. Section 2 discusses three practical erase suspension mechanisms: I-ES, D-ES, and T-ES. We add the following three configurations to our evaluation for comparison:

- Baseline: Erase operations do not get preempted by an incoming read request.
- Erase Suspension (*ES*): The scheme can suspend and resume an erase pulse from an arbitrary point as proposed by Wu et al. [35].
- Ideal Erase Suspension (*Ideal-ES*): This scheme can suspend and resume an erase operation from any arbitrary point with *zero* erase suspension penalty.

3.2 Random Access Benchmark

Workload. We first evaluate our erase suspension policies using a microbenchmark that generates a mixture of 4KB random reads (70%) and writes (30%) at queue depth 16. We utilize Flexible I/O Tester (FIO) [8] to generate such disk access patterns. This workload is widely used to evaluate an SSD's latency performance [10, 38]. Figure 4 shows the results of this experiment.

Read Tail Latency. In the baseline, when an erase happens,

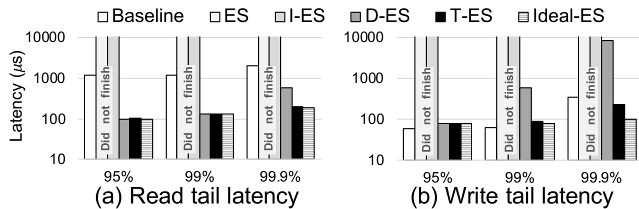


Figure 5: Read/Write tail latency on ACT workload (30× workload multiplier).

Table 3: Performance and stress test results using ACT.

	Baseline	ES	I-ES	D-ES	T-ES	Ideal-ES
Stress	32×	22×	22×	30×	30×	32×
Performance	14×	22×	22×	30×	30×	32×

the subsequent read requests are simply delayed for the duration of the remaining erase time (up to 5ms). As a result, baseline policy shows around 5ms read tail latency. With ES and I-ES such an erase operation is repeatedly aborted by the incoming read requests. As a result, incoming read requests do not experience an extra delay caused by the ongoing erase operation except for the 100μs latency required for erase suspension/resumption. For this reason, both ES and I-ES policies have very low read tail latency. Note that they are prone to experience erase (and write) starvation as pointed out in Section 2.2. However, it does not happen here because this workload has a period when read requests are not sent out for some time (i.e., all 16 outstanding requests in the queue are write requests waiting for an erase to proceed). During this period, erase operation successfully finishes. The read tail latency of D-ES is around 1ms because it always lets the ongoing erase step (not the whole erase) finish, making a read request wait. T-ES behaves similarly to I-ES in this workload as the timeout is rarely triggered.

Write Tail Latency. Baseline, ES, D-ES, and Ideal-ES have a lower write tail latency because their erase operation is finished in a relatively short period of time without canceling an existing erase pulse. Notably, the write tail latency of the baseline policy is the maximum GC latency (35ms) of our model without erase suspension. On the other hand, both I-ES and T-ES abort the erase multiple times, and thus delays write operations significantly. As a result, they tend to have a longer write tail latency.

3.3 Database Benchmark

Workload. ACT models the Aerospike database servers' real-time I/O access patterns. In essence, ACT consists of three threads: one issuing 2K small (1.5KB) read requests per second, another issuing 24 large (128KB) read requests per second, and the third one issuing 24 large (128KB) write requests per second. ACT gradually increases this rate in integer multiples (e.g., ACT 4× workload means 8K small read requests/s, 96 large read requests/s, 96 large write requests/s) and considers that the device has passed the *performance test* if it meets the following conditions for a long time (e.g., 24hrs): i) 95% of transactions finish in 1ms, ii) 99% of transactions finish in

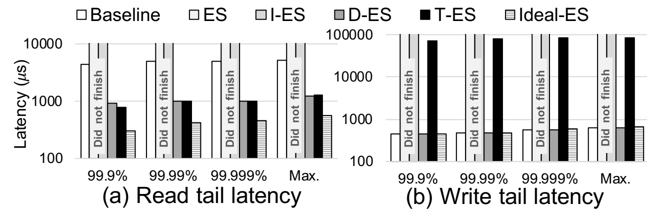


Figure 6: Read/Write tail latency on TPC-C.

8ms, iii) 99.9% of transactions finish in 64ms, and iv) average transaction time for each type of requests is smaller than ACT workload's I/O request period. If a device satisfies only the fourth condition but fails to satisfy one of the first three conditions, that device is said to pass the *stress test* but not the *performance test*. The maximum ACT multiplier that an SSD can satisfy is that particular SSD's performance rating.

ACT Results. Table 3 shows the maximum multiplier in which each configuration passed the performance and stress tests. Figure 5 shows the tail latency of 95%, 99%, and 99.9% accesses at 30× workload multiplier. As shown in Table 3, the baseline has a good average response time and thus can successfully run ACT with 32× multiplier. On the other hand, it has the worst read tail latency since the part of erase latency (up to 5ms) is exposed to read requests. This leads to a relatively poor performance test result for the baseline. ES and I-ES have good read tail latency behavior. However, continuous read requests cause erase (and write) starvation, which leads to a failure in the stress test (fourth condition) at the workload multiplier whose value is above 22×. On the other hand, both D-ES and T-ES demonstrate strong results for both stress and performance tests. Both D-ES and T-ES can maintain low read tail latency while avoiding erase (and write) starvation.

3.4 Transaction Processing Benchmark

Workload. TPC-C [34] is a popular benchmark for online transaction processing frameworks. We utilize a published disk trace of the system running TPC-C from SNIA [31], to evaluate our erase suspension mechanisms.

TPC-C Results. Figure 6 shows the results from this experiment. The baseline policy generally observes a noticeably higher read tail latency than the other schemes since latency is exposed to many read requests queued behind the erase operation. Neither ES or I-ES runs to completion as they suffer from a severe erase (and write) starvation and cause the simulator to exit prematurely after failing to serve many write requests for a long time. As in the ACT workload, this is because continuous read requests prevent the completion of an erase operation. Both T-ES and D-ES achieve much lower tail latency than the baseline. In particular, both achieve around 1ms max tail latency, which indicates that a read request only experiences about a single erase step delay at most.

The write tail latency of the baseline, D-ES, and Ideal-ES is similar to each other. In contrast, ES and I-ES suffer write starvation, and T-ES records the longest write tail latency. The

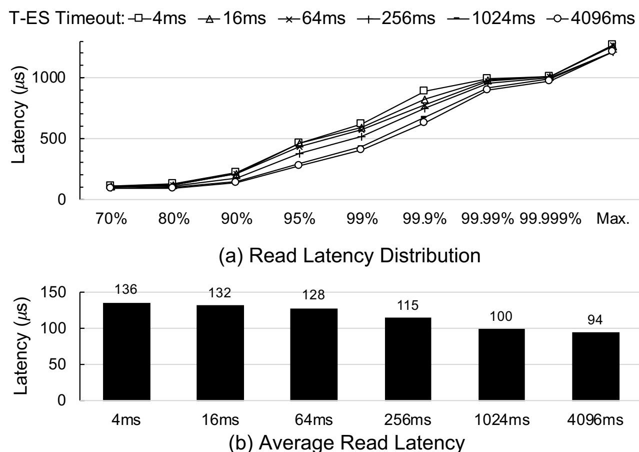


Figure 7: (a) Read latency distribution and (b) average read latency with varying T-ES timeout values for TPC-C.

Table 4: Maximum write latency for TPC-C.

T-ES Timeout	4ms	16ms	64ms	256ms	1.02s	4.09s
Write Latency (Maximum)	27ms	40ms	86ms	280ms	1.04s	4.1s

T-ES scheme delays erase (and following writes) for up to the timeout value hoping to find a period in which it can perform an erase without blocking reads. In this case, T-ES does not find such a period and hence ends up triggering the D-ES mechanism after delaying the write requests; as expected, the maximum write latency converges within 90ms (i.e., the sum of GC latency (24ms) and the T-ES timeout value (64ms)).

3.5 Sensitivity to T-ES Timeout Threshold (N)

If an erase operation is delayed for Nms, T-ES switches from I-ES to D-ES to avoid erase (and hence write) starvation. We perform a sensitivity study, using TPC-C with varying N to provide an insight into the tradeoff with selection of this parameter.

Figure 7(a) shows the read latency distribution with different values of N from 4ms to 4096ms. At around the 80th percentile we start to observe a gradual transition of read latency from more of I-ES (about 200μs) to D-ES (about 1ms). Increasing N generally i) lowers frequency of both high-latency read (i.e., over-200μs) and ii) average read latency, but iii) increases the maximum write latency. As N increases, T-ES has a greater chance of running in I-ES mode to lower the chance for a read request to experience a 1ms delay for finishing an ongoing erase pulse. Fewer long-latency reads lead to a lower average read latency as shown in Figure 7(b).

However, increasing N negatively affects the maximum write latency. Table 4 summarizes the maximum write latency with varying N. As discussed in Section 2.4, the maximum write latency of T-ES is the sum of GC latency and the T-ES timeout value. This is because GC operations, which need an erase operation to produce a free block for user data write, may be interfered while running in I-ES mode. Once T-ES timeout is triggered, it switches to D-ES to allow GC operations to produce the free blocks to write user's data. The maximum GC latency of the TPC-C workload (with steady state pre-condition) is 24ms. Thus, the measured maximum write latency is not far off from the estimated value (i.e., GC latency plus N).

4 Related Work

Garbage Collection Optimization. There are several prior works on alleviating the effect of GC on tail latency [6, 13–15, 17, 37]. While these optimizations can effectively reduce the effect of coarse-grained GCs on read tail latency, they do not address the effect of long erase operation on read tail latency, which becomes more important with optimized GC. Thus, these techniques are orthogonal or complementary to our presented erase suspension schemes.

I/O Scheduling Optimization. Another way to optimize tail latency is to schedule a read/write request in an intelligent way [16, 20, 21, 36, 39, 40]. These proposals are effective when there are multiple devices available. On the other hand, our work focuses on tail latency reduction without requiring multiple devices. Still, if multiple devices are available, our technique can be applied jointly with these optimizations.

5 Conclusion

This paper introduces practical erase suspension mechanisms to limit the impact of erase operation on long read tail latency. Leveraging the iterative erase mechanism commonly employed by today's flash devices, the proposed mechanisms minimize the impact of erase operation on read tail latency, while requiring only minor extensions to the flash interface. With the proposed erase suspension mechanisms, our proposal enables flash-based SSDs to achieve very low read tail latency, while avoiding erase (and write) starvation and endurance degradation of NAND. For example, our results demonstrate the feasibility of a sub-200μs 99.999th percentile read tail latency for 4KB random access workloads, which is competitive with an emerging non-flash NVM-based SSD [10]. This will harness the full potential of flash-based SSDs as the primary storage platform for future data-centers that will be required to run a variety of latency-sensitive online services.

Acknowledgments

We thank Sam H. Noh for shepherding this paper and the anonymous reviewers for their feedback. We also thank Adel Choi, Heesoo Kim, Donghyeon Kwon, and Daejoong Jung for their support. This work was supported by Research Resettlement Fund for the new faculty of Seoul National University, a research grant from Samsung Electronics, and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea Government (MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore OS). Jae W. Lee is the corresponding author.

References

[1] Aerospike Certification Tool. https://github.com/aerospike/act.

[2] 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):Morgan & Claypool Publishers, 1–154, 2013.

[3] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the IEEE 30th International Conference on Computer Design*, ICCD'12, pages 94–101. IEEE, 2012.

[4] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3(4):ACM, 837–863, November 2004.

[5] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel DG Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *Proceedings of the IEEE International Solid-State Circuits Conference*, ISSCC '18, pages 338–340. IEEE, 2018.

[6] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. Parallelizing garbage collection with I/O to improve flash resource utilization. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 243–254. ACM, 2018.

[7] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):ACM, 74–80, 2013.

[8] Flexible I/O Tester. https://github.com/axboe/fio.

[9] Alan Freedman. MT/sec. *The Computer Desktop Encyclopedia*. https://www.computerlanguage.com/results.php?definition=MT/sec.

[10] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):IEEE, 1822–1833, 2017.

[11] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST '16, pages 263–276. USENIX Association, 2016.

[12] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 61–74. USENIX Association, 2014.

[13] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. HIOS: A host interface I/O scheduler for solid state disks. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture*, ISCA '14, pages 289–300. ACM/IEEE, 2014.

[14] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. *ACM Transactions on Embedded Computing Systems*, 16(5s):ACM, 134:1–134:20, 2017.

[15] Wonkyung Kang and Sungjoo Yoo. Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 8:1–8:6. ACM, 2018.

[16] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an autonomic SSD architecture. In *Proceedings of the USENIX Annual Technical Conference*, ATC'18, pages 677–690. USENIX Association, 2018.

[17] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO complying SSDs through OPS isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 183–189. USENIX Association, 2015.

[18] Dong Wook Lee, Sunghoon Cho, Byung Woo Kang, Sukkwang Park, Byoungjun Park, Myoung Kwan Cho, Kun-Ok Ahn, Ye Seok Yang, and Sung Wook Park. The operation algorithm for improving the reliability of TLC (triple level cell) NAND flash characteristics. In *Proceedings of the 3rd IEEE International Memory Workshop*, pages 1–2. IEEE, 2011.

[19] Jaeduk Lee, Jaehoon Jang, Junhee Lim, Yu Gyun Shin, Kyupil Lee, and Eunseung Jung. A new ruler on the storage market: 3D-nand flash for high-density memory and its technology evolutions and challenges on the future. In *Proceeding of the 2016 IEEE International Electron Devices Meeting (IEDM)*, pages 11.2.1–11.2.4, Dec 2016.

[20] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):IEEE, 247–260, 2013.

[21] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST'03, pages 131–144. USENIX Association, 2003.

[22] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? When? and Why? In *Proceedings of the 9th ACM International Systems and Storage Conference*, SYSTOR'16, pages 7:1–7:11. ACM, 2016.

[23] NVM Express 1.3a. http://nvmexpress.org/.

[24] Krishna Parat and Chuck Dennison. A floating gate based 3D NAND technology with CMOS under array. In *Proceeding of the 2015 IEEE International Electron Devices Meeting (IEDM)*, pages 3.3.1–3.3.4, Dec 2015.

[25] Byoungjun Park, Sunghoon Cho, Milim Park, Sukkwang Park, Yunbong Lee, Myoung Kwan Cho, Kun-Ok Ahn, Gihyun Bae, and Sungwook Park. Challenges and limitations of NAND flash memory devices based on floating gates. In *Proceeding of the 2012 IEEE International Symposium on Circuits and Systems*, pages 420–423, 2012.

[26] Jisung Park, Jaehoon Lee, Myungsuk Kim, Myungjun Chun, and Jihong Kim. Reducing read latency fluctuations of flash storage systems using preemptible programs and erases. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies, Work-in-Progress Reports (WiPs)*, FAST '18. USENIX Association, 2018.

[27] Sung-Kye Park. Technology scaling challenge and future prospects of DRAM and NAND flash memory. In *Proceeding of the 2015 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2015.

[28] PCI Express 3.1. https://pcisig.com/specifications/.

[29] C. Sandhya, Apoorva B. Oak, Nihit Chattar, Udayan Ganguly, C. Olsen, S. M. Seutter, L. Date, R. Hung, Juzer Vasi, and Souvik Mahapatra. Study of P/E cycling endurance induced degradation in SANOS memories under NAND (FN/FN) operation. *IEEE Transactions on Electron Devices*, 57(7):1548–1558, July 2010.

[30] Kent Smith. Understainding SSD overprovisioning. In *Proceedings of the Flash Memory Summit (2012)*, Flash Memory Summit'12, 2012.

[31] SNIA IOTTA repository. TPC-C traces. http://iotta.snia.org/traces/131.

[32] SNIA Solid State Storage Performance Test Specification. https://www.snia.org/sites/default/files/HoEasen_SNIA_Solid_State_Storage_Per_Test_1_0.pdf.

[33] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 49–66. USENIX Association, 2018.

[34] TPC-C benchmark. http://www.tpc.org/tpcc/.

[35] Guanying Wu and Xubin He. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 117–123. USENIX Association, 2012.

[36] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware request steering with improved performance and reliability for SSD-based RAIDs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS'18, pages 296–305. IEEE, 2018.

[37] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 15–28. USENIX Association, 2017.

[38] Samsung Z-SSD SZ985. https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[39] Jianyong Zhang, Alma Riska, Anand Sivasubramaniam, Qian Wang, and Erik Riedel. Storage performance virtualization via throughput and latency control. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS'05, pages 135–142. IEEE, 2005.

[40] Quan Zhang, Dan Feng, Fang Wang, and Yanwen Xie. An efficient, QoS-aware I/O scheduler for solid state drive. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, HPCC'13, pages 1408–1415. IEEE, 2013.

Track-based Translation Layers for Interlaced Magnetic Recording

Mohammad Hossein Hajkazemi⋆, Ajay Narayan Kulkarni†, Peter Desnoyers⋆, Timothy R Feldman†
Northeastern University⋆, Seagate Technology†

Abstract

Interlaced magnetic recording (IMR) is a state-of-the-art recording technology for hard drives that makes use of heat-assisted magnetic recording (HAMR) and track overlap to offer higher capacity than conventional and shingled magnetic recording (CMR and SMR). It carries a set of write constraints that differ from those in SMR: "bottom" (e.g. even-numbered) tracks cannot be written without data loss on the adjoining "top" (e.g. odd-numbered) ones. Previously described algorithms for writing arbitrary (i.e. bottom) sectors on IMR are in some cases poorly characterized, and are either slow or require more memory than is available within the constrained disk controller environment.

We provide the first accurate performance analysis of the simple *read-modify-write (RMW)* approach to IMR bottom track writes, noting several inaccuracies in earlier descriptions of its performance, and evaluate it for latency, throughput and I/O amplification on real-world traces. In addition we propose three novel memory-efficient, track-based translation layers for IMR—*track flipping*, *selective track caching* and *dynamic track mapping*, which reduce bottom track writes by moving hot data to top tracks and cold data to bottom ones in different ways. We again provide a detailed performance analysis using simulations based on real-world traces.

We find that RMW performance is poor on most traces and worse on others. The proposed approaches perform much better, especially dynamic track mapping, with low write amplification and latency comparable to CMR for many traces.

1 Introduction

Magnetic recording technology has made enormous strides over the last several decades, reaching densities of about a terabit per square inch, higher than that of any but the most modern and densest solid-state storage technologies. Yet in recent years density improvements have run up against the *superparamagnetic limit* [17]—as bits get smaller, the magnetic media coercivity (resistance to being magnetized) must go up, to avoid bit flips from thermal noise, while as heads get smaller their magnetic field becomes weaker, requiring lower coercivity media. In other words, smaller bits require smaller track sizes, requiring smaller write heads, requiring lower-coercivity media, resulting in larger minimum bit sizes. When the minimum bit size becomes as large as the write head, further density improvements require new approaches. We are currently at or near this limit; increases in disk capacity in the past 5 years or more have relied more on increasing the number of platters per drive rather than increases in areal density.

New strategies allow further increases in areal density by sidestepping one or both sides of this trade-off, i.e. either breaking the link between bit size and write head size / magnetic field strength, or between bit reliability and media coercivity. Shingled Magnetic Recording (SMR) [1] overlaps adjacent tracks, reducing the effective track width without reducing the write head size. Yet, this increase in density comes at a cost: random writes are not allowed, as overwriting a sector will also overwrite the corresponding sector in the adjacent "downstream" track, and therefore the data could be lost. Heat-Assisted Magnetic Recording [12] takes advantage of the fact that the coercivity of a material goes down with temperature, and uses a laser to heat the media to near the Curie point[1] before writing. This allows use of a medium with much higher room-temperature coercivity and smaller grain size (and thus minimum bit size), and also allows an effective track width narrower than the write head by narrowing the width of the heated domain, by controlling the laser current.

Interlaced Magnetic Recording (IMR) [9] uses both heat-assisted recording and track overlap. This is in contrast to SMR, which uses track overlap but conventional room-temperature recording. As shown in Figure 1, tracks are written in an "interlaced" fashion, with a "bottom" layer of tracks written first, after which a "top" layer is written between (and partly overlapping) these bottom tracks. To avoid total overwrite of the bottom tracks, top tracks are written with a narrower width and thus slightly lower capacity, roughly 90% that of the

[1]The temperature above which the material will no longer retain its magnetic properties.

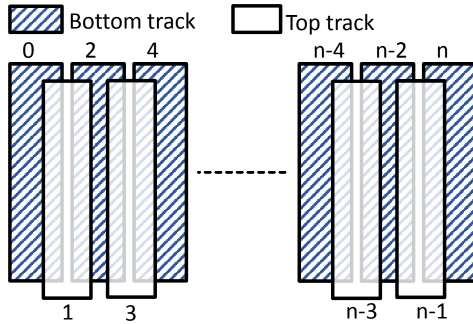


Figure 1: IMR technology: tracks are written in an interlaced fashion; top tracks are written between and over bottom tracks. Top track *i* partially overlaps bottom track *i-1* and *i+1*.

bottom tracks. The result is a drive with write constraints, but ones that are far less strict (and thus less performance-limiting) than those for SMR. Where SMR writes must be limited to a single track per zone (a group of a few hundred of tracks) to avoid data loss, in IMR nearly half of the sectors (i.e. those on the top tracks) can be re-written safely. Moreover whereas moving valid data in SMR requires reading and/or writing an entire (typically 256 MB) zone, for IMR in the worst case only two tracks (less than 5 MB) must be moved.

IMR is a very new technology, with the first descriptions of its physical feasibility dating to 2016 [4,9]; with one exception, Wu et al. [20] that is discussed in Section 5 the publications to date have focused on the magnetic and physical aspects of IMR, rather than system implications and algorithms. We provide the first thorough performance analysis of the naive *read-modify-write (RMW)* strategy described in the first IMR proposals [4], correcting several mistaken assumptions, and quantifying the performance degradation of IMR with RMW for real workloads. We offer three algorithms for improved management of IMR writes, *track flipping*, *selective track caching* and *dynamic track mapping*, all of which (unlike the approach of Wu et al.) may be readily implemented in disk firmware with limited memory and compute resources. We provide detailed performance models of these algorithms, and evaluate them in simulation on real workloads, showing substantial improvement over RMW in all cases and near-conventional-drive performance for some workloads.

In particular, the contributions of this paper are:

1. a thorough performance analysis of naive read-modify-write for IMR disk—i.e. as proposed in Hwang et al. [9]—showing a performance overhead of more than 2x that assumed by prior work,

2. three novel track-based translation layers—track flipping, selective track caching and dynamic track mapping—which mitigate most of the IMR performance penalty at a sufficiently modest cost in memory that they may be implemented within today's on-board disk controllers,

3. evaluation of conventional ("CMR") disk, RMW and the three proposed algorithms on real-world traces, demonstrating (a) significant costs to RMW vs. CMR, and (b) substantial improvements for all proposed algorithms particularly dynamic track mapping.

2 Algorithms for IMR

As with SMR, IMR write limitations maybe addressed from the host or within the device; however the complexity of the IMR track-to-track write restrictions makes it preferable to employ a device-based block translation layer rather than expose restrictions to the host.

We describe in detail four algorithms: naive read-modify-write [9], track flipping, selective track caching and dynamic track mapping. For each algorithm we estimate memory usage, describe the data copies and logging of mapping changes needed to prevent data loss in the case of a crash, and analyze the performance of the algorithm's operations.

2.1 Read-modify-write

The simplest IMR translation layer is what we term naive read-modify-write (RMW). Sectors on disk are assigned fixed logical addresses, as in conventional drives, and before performing any write to some sector S on a bottom track T, the drive (1) reads the adjacent top track sectors (S_{T-1} on $T-1$ and S_{T+1} on $T+1$) and (2) copies them to a "backup region", then (3) performs the bottom-track write, and finally (4) re-writes the adjacent top track sectors.

When numbering tracks $T-1$, T, $T+1$ we are referring to physical position, which may not directly correspond to logical block numbering. In particular IMR is expected to use a *serpentine* or *zig-zag* layout [10], where LBAs are numbered sequentially across N adjacent bottom tracks, and then across the corresponding N top tracks. The result is that sectors in physically adjacent top and bottom tracks will be separated by a distance of N track sizes either in all cases (zig-zag) or on average (serpentine).

Memory usage: Other than buffers for copying, no additional memory is required beyond that needed for standard LBA to physical location translation in a conventional drive.

Safety and crash consistency: We first note that to avoid data loss, host writes to the affected top tracks must be blocked during the RMW operation, as they would be overwritten when data is copied back to the tracks. The duration of this locking determines a phase, which is atomic with respect to user I/O, and mapping updates need only be persisted once per phase.

Copying sectors S_{T-1} and S_{T+1} to the backup region forms a single phase, and the temporary location of the sectors is logged to the backup region just before the phase completes. If a crash occurs before restoring the top tracks, a startup scan of the log will locate the saved data, which may be copied back to its proper location. The length of the log is determined by the number of simultaneous RMW operations allowed; if this

is 1, then no log trimming is needed as it will just be replaced by the log from the next operation.

When logging data to the backup region, we can write additional metadata with negligible overhead, much like journal entries in a file system. Efficiently persisting the fact that S_{T-1} and S_{T+1} have been restored is more difficult, however; if this is not done, then future writes to these locations may be lost if stale backup data is copied back on restart. A straightforward way to do this is to clear the backup region; however this requires an additional seek and possibly lost rotation. Instead we clear the backup region *lazily*, if we detect a write to S_{T-1} or S_{T+1}. Since *any* RMW operation will clear the previous contents of the backup region, in most cases this lazy cleaning may be omitted, as until S_{T-1} or S_{T+1} are modified, the backup data is not stale and may be copied back safely on startup.

Timing: The performance of this approach may be analyzed by examining the steps above. We assume a random single-sector write to sector S on bottom track T, and assume as well that sectors S_{T-1} and S_{T+1} on tracks $T-1$ and $T+1$ respectively must be moved to avoid data loss. The time taken is thus at least:

1. $0.5t_{rot} + t_{seek}$ to reach and read sector S_{T-1}, where t_{rot} is the rotation time, assuming an average 0.5-rotation delay for random access.
2. A missed rotation, t_{rot}, to reach and read sector S_{T+1}.
3. $t_{seek} + 0.5t_{rot}$ to reach the backup region, plus negligible transfer time to write sector S_{T-1} and S_{T+1}.
4. $t_{seek} + 0.5t_{rot}$ (see below) to reach and write sector S on track T.
5. a missed rotation (t_{rot}) plus negligible transfer time to reach and write S_{T-1} on track $T-1$.
6. a missed rotation (t_{rot}) plus negligible transfer time to reach and write S_{T+1} on track $T+1$.

Steps 3 and 4 together will take an integral number of rotations, either 1 or 2, depending on whether the disk is able to seek to the safe track, wait until the write location passes under the head, and seek back within a single rotation (t_{rot}). Based on discussions with disk vendors we assume the two steps will take $2t_{rot}$ to complete, for a random write latency of $t_{seek} + 5.5t_{rot}$. The same operation would take $t_{seek} + 0.5t_{rot}$ on a CMR drive, for a RMW overhead of 5 rotations

Performance is even worse for sequential write, as the previous write finishes just after writing sector S, so that step 1 will require an entire missed rotation, for a total latency of $6t_{rot}$. Note that multiple writes to the same track may be coalesced into a single RMW operation, whether via command queuing or the use of write caching on the drive; however it will still take 5 or 6 rotations longer than writing a full track on a conventional drive.

We note that in our analysis, the actual performance of RMW will be significantly worse than that implied by Hwang et al. [9], where they state that writes to bottom tracks will require two rewrites, for a mean of one extra rewrite per host write request. We attribute the inaccuracy of their analysis[2] to several factors: (1) the significance of missed rotations in the rewrite process, each of which is far more costly than all but the largest write requests; (2) the need to read top-track data so that it can be re-written, and (3) the need to persist top-track data in a secondary location, to avoid data loss from failure in the middle of a RMW operation.

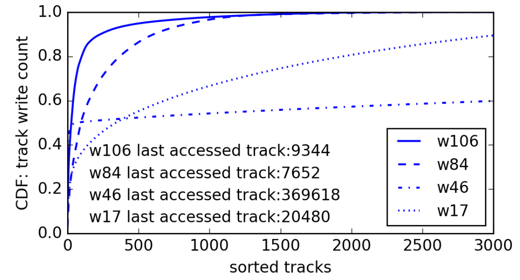


Figure 2: Track write count CDF (the first 3000 hottest tracks) for traces w17, w46, w84, w106 (See Section 3 for trace description); tracks are sorted from the hottest (i.e. track 0) to the coldest (i.e. track 3000).

2.2 Track flipping

Real workloads show high locality, with typically a small number of *hot* sectors being overwritten frequently, and the remaining sectors receiving few if any writes; the same phenomena is found at the track level, as shown in Figure 2 (an illustration of the first 3000 hottest tracks in a few workloads). For instance, a significant portion of writes (80%) are received by a small number (100) of tracks in w106. We can take advantage of this locality by moving data between tracks to maximize the amount of hot data stored on re-writable top tracks. Our first algorithm, *track flipping*, locates bottom tracks containing hot sectors (i.e. *hot tracks*) and swaps them with adjacent top tracks, moving the hot data to the top, where additional writes can be performed directly, and (hopefully) moving cold data to the bottom track. In particular, we track the number of writes to each track, periodically identify candidates for flipping—i.e. hot bottom tracks which are adjacent to cold top tracks—and swap them. The actual swap of tracks T (bottom) and $T+1$ (top) is straightforward:

1. read tracks $T-1$, T, and $T+1$
2. write $T-1$ and T contents to a backup region[3]
3. write $T+1$ contents to T
4. write T contents to $T+1$
5. rewrite $T-1$

Implementation of this algorithm must take into account several real-world factors. Top and bottom tracks hold

[2] To be fair, their analysis is a minor paragraph in the middle of a magnetics paper.

[3] The backup region must accommodate at least two tracks.

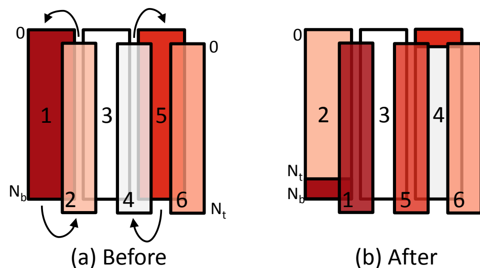


Figure 3: Track flipping: hot bottom tracks 1 and 5 (red) are swapped with cold top tracks 2 and 4. Since top tracks (N_t sectors) are smaller than bottom tracks (N_b sectors), only the first or last N_t sectors of hot bottom tracks are moved.

differing amounts of data, with top tracks estimated to have 90% the capacity of bottom tracks. In addition, neighboring top tracks or bottom tracks may vary slightly in capacity, due to the presence of bad sectors hidden by *slip sparing* [10]—i.e. the LBA numbering skips a bad sector, resulting in a track containing fewer sectors than if it were perfect. We note that there are other variations in track capacity due to the use of zone bit recording [10] and adaptive formatting [11]; however in all but a negligible number of cases these will not result in differing capacities for adjacent tracks.

Our solution to differing track capacities is to swap *most* of the bottom track with the top track contents, as shown in Figure 3. If a bottom and adjacent top track hold N_b and N_t sectors respectively, then we can swap the first N_t sectors of the bottom track with the entire contents of the top track (tracks 4 and 5 in Figure 3). In the case where hot sectors are located at the "end" of the bottom track, we instead swap the last N_t sectors of the bottom track with the contents of the top track (tracks 1 and 2 Figure 3). Given the original location of a sector (i.e. sector position S on track T) the sector can be located precisely in the flipped configuration given knowledge of which flip (low LBA or high LBA) has been performed and the exact track sizes N_t and N_b, which are already known by the firmware as part of the LBA translation process. Note that once two tracks have been flipped, data cannot migrate any further; if tracks T and $T+1$ have been flipped, then flipping $T-1$ and T, or $T+1$ and $T+2$, is not allowed until T and $T+1$ have been flipped back.

Memory usage: The track mapping may be represented in a very concise fashion, as each bottom track T is in one of five states: (1) unmoved, (2) its low LBAs flipped with track $T-1$, (3) its high LBAs flipped with $T-1$, or (4) and (5), its high or low LBAs flipped with $T+1$. The resulting track map requires 3 bits per bottom track, or 1.5 bits per track; assuming a mean track size of 1.5 MB, this would require a map of about 2.5 MB for a 20 TB drive.

Memory requirements for hot track detection can be modest, as well. The total number of tracks is large, 1.3×10^7 for our 20 TB drive; however the number of tracks written in the period between iterations of the track flipping algorithm is much smaller (e.g. 20K in our experiments). Logging these track numbers in memory (using data structures such as an array or a list) will take less than 0.25 MB, and they may then be sorted and counted to determine track write frequency during that interval.

Safety and crash consistency: For track flipping we need to persist not only the state of the flipping process, but also updates to the track mapping. The flip process involves one more copy than RMW, but may be handled in the same way, by keeping an update log in the backup region, and marking sectors when they are copied back to their home (or flipped) locations. Backup region metadata can include a small log of map updates which can be appended to the track map in batches. To persist changes to the track map we keep a copy of the map on disk, and a log of updates to the map in the "checkpoint location". The checkpoint can be rewritten periodically and the log recycled, resulting in a negligible amortized cost for persisting map changes.

Timing: Assuming the head starts in an arbitrary location, the time required to flip bottom track T and top track T+1 will be:

1. t_{seek} to reach track T
2. $3t_{rot}$ to read tracks $T-1$, T and $T+1$[4]
3. t_{seek} to reach a backup region
4. $2t_{rot}$ to write backup copies of track $T-1$ and T
5. t_{seek} to return to track T
6. t_{rot} to write the contents of $T+1$ into track T
7. t_{rot} to write the contents of track T into $T+1$
8. t_{rot} to rewrite the contents of track T-1

for a total cost of $3t_{seek} + 8t_{rot}$. Since all accesses are to entire tracks, we assume that existing disk scheduling and buffering mechanisms allow reading or writing to begin immediately after reaching a track, rather than incurring additional rotational delay. Note, however, that track flipping is a background operation, and can be interrupted at any point in time—resuming an interrupted flip is very similar to the crash recovery scenario, except that in-memory state is still available. The primary performance impacts of track flipping are thus a reduction in overall throughput, from the background flipping process, in combination with RMW latency for those bottom-track writes to tracks which have not been flipped.

For track flipping to be effective, hot bottom tracks must be paired with neighboring cold top tracks, as there would be no advantage to flipping the two tracks of the same "temperature". Although one can easily construct synthetic workloads (e.g. uniform random) which lack neighboring hot/cold track pairs, we wish to determine whether they are found in real-world workloads. To address this question we analyze one of our experimental workloads (w17, described in the Section 3 below), assuming a constant track size of 2 MB. In Figure 4 we see write counts for the 20 hottest tracks and their neighbors.

[4] Seeks due to track switches as well as short seeks from track $T-1$ to track $T+1$ and vice versa are not included in our calculations.

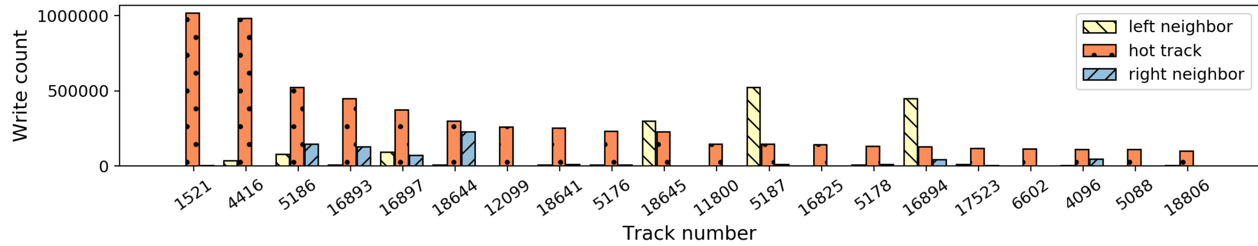


Figure 4: Write count of 20 hottest tracks and their neighbors, trace w17. This workload is seen to be “track flipping-friendly”.

In only a few cases (e.g. track 5187) do hot tracks have a hot neighbor; however even in those cases the other neighbor is cold. Similar results are seen in many—but not all—other workload traces. However we note that results may vary with file systems other than the ones found in our traces, i.e. ext4 and NTFS, and will certainly vary with differing track sizes.

Real-world workloads are time-varying, with the identity of hot locations changing over time. We see this in Figure 5, which shows the write frequency over time for a range of 4 tracks. Not only does write frequency to a given track vary, but relative write frequency between tracks changes as well: e.g. track 3854 is much hotter than 3857 for a significant period, while later in the trace track 3857 is hotter. By using time-limited write counts, which are periodically reset after each search for hot tracks to flip, we are able to adapt to these changes in access frequency. In Algorithm 1 we see the full track-flipping algorithm: every N writes (e.g. 20,000) we select the hottest *k* bottom tracks over the last interval and, if possible, switch them with cold neighbors.

Algorithm 1: track flipping

```
   parameter : updateFrequency, flipThreshold, maxFlips
   variable  : ioDirection (read/write), trackPosition
               (bottom/top), trackNumber, writeCount,
               flipCount, trackIdLog [], trackCounts []
 1 ioDirection, trackPosition, trackNumber ← ReceiveIO()
 2 if ioDirection == write then
 3   writeCount ++
 4   trackIdLog.append(trackNumber)
 5   if writeCount mod updateFrequency == 0 then
 6     trackCounts [] ← Count (trackIdLog)
 7     for every track in Hottest (trackCounts) do
 8       flipCount ++
 9       middleCounter = trackCounts [track ]
10       leftCounter = trackCounts [track-1]
11       rightCounter = trackCounts [track +1]
12       selected = Min (leftCounter, rightCounter)
13       temperatureDiff = middleCounter- selected
14       if temperatureDiff > flipThreshold then
15         TrackFlip (track, selected) maxFlips ++
16       end
17       if flipCount >= maxFlips then
18         Break ()
19       end
20     end
21   end
22 end
```

2.3 Selective track caching

With track flipping—as with RMW—every track on the disk except for the two “backup region” tracks is filled with user data, requiring significant “data shuffling” to move data. If we instead reserve a small number of tracks for translation layer use, we can achieve additional gains in performance. *Selective track caching* does precisely this, reserving a small range of non-interlaced bottom-only tracks as a persistent cache for holding data from hot bottom tracks. Whereas track flipping is not able to move a hot bottom track if both of its neighbors are also hot or if both of its neighbors are already flipped with another bottom track, selective track caching is able to move any hot bottom track, at any time.

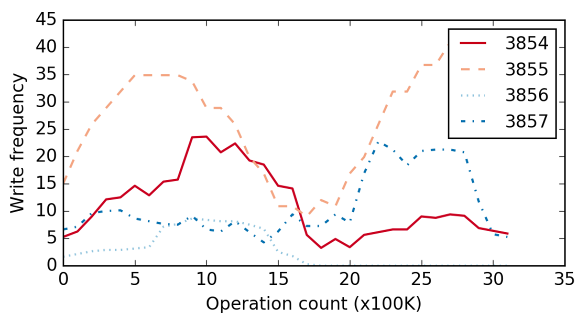


Figure 5: Track write frequency for tracks 3054-3057, trace w106. Y axis is the number of writes to a track out of 100,000 total writes.

More specifically, we reserve *k* bottom tracks as a random-write region (much like an SMR persistent cache), either at the outer diameter to maximize track size and transfer rate, or distributed in smaller groups across the disk to minimize seek time to the nearest cache. We note that tracks in the persistent cache will not be precisely the size of tracks that are cached there; instead we allocate “logical tracks” within the cache, where each logical track is a range of LBAs long enough to

hold a full track and a metadata header. As seen in Algorithm 2, we again monitor track write counts, and periodically select the hottest bottom tracks to be moved to persistent cache, while moving the coldest cached tracks back to their home location.

Memory usage: For this algorithm, memory is needed for monitoring track write counts and for keeping a map of the cached tracks. The requirements for track write monitoring are the same as they are for track flipping, and thus the same approaches may be used with identical memory usage: hundreds of KB for logging the track numbers written in the period between iterations, or negligible usage if write tracking already performed by the drive is adequate. If the cache map is structured as a look-aside list of exceptions to the standard map, then its memory usage is proportional to the size of the persistent cache, not the drive itself. In our experiments a cache of 100 tracks was used, requiring a trivial amount of memory; however for a cache of several tens of thousands of tracks, memory usage should still remain in the range of a few MB.

Safety and crash consistency: The same approach may be used for maintaining a consistent copy of the map as for track flipping: updates are logged, and a full checkpoint written periodically. Alternately if the cache size is sufficiently small we can exhaustively scan the cache and rebuild the map on startup; however this requires t_{rot} per track, and starts to become impractical at cache sizes of less than 100 tracks.

Timing: The actual data movement portion of this algorithm is straightforward. Promotion of a bottom track to the cache merely requires seeking to it (t_{seek}), reading it (t_{rot}), seeking to the cache (t_{seek}), and writing it (t_{rot}), for a total of $2t_{seek} + 2t_{rot}$. However track eviction takes longer as it requires a full-track RMW operation; the time taken will be:

1. t_{seek} to reach track C in the cache
2. t_{rot} to read track C
3. t_{seek} to reach track $T-1$
4. $2t_{rot}$ to read track $T-1$ and $T+1$
5. t_{seek} to reach to the backup region
6. $2t_{rot}$ to write backup copies of track $T-1$ and $T+1$
7. t_{seek} to seek back to track T
8. t_{rot} to write the contents of track C into T
9. $2t_{rot}$ to re-write track $T-1$ and $T+1$

Algorithm 2: selective track caching

```
   parameter : updateFrequency, cacheSize
   variable  : ioDirection, trackPosition,
               trackNumber, writeCount, trackIdLog
               [], cachedTracks [], trackCounts [], victim
1  ioDirection, trackPosition, trackNumber ← ReceiveIO()
2  if ioDirection == write then
3  |  writeCount ++
4  |  trackIdLog.append(trackNumber)
5  |  if writeCount mod updateFrequency == 0 then
6  |  |  trackCounts [] ← Count (trackIdLog)
7  |  |  for every track in Hottest (trackCounts) do
8  |  |  |  if track not in cache then
9  |  |  |  |  victim ← Coldest (cachedTracks)
10 |  |  |  |  TrackSwap (track, victim)
11 |  |  |  end
12 |  |  end
13 |  end
14 end
```

for a total cost of $4t_{seek}$+$8t_{rot}$. In steady state one track will be evicted for every track promoted, for a total cost of $7t_{seek} + 8t_{rot}$ per track promoted. Batching of promotions and evictions may remove several seek times from this total, but will not make great improvements due to the scattered locations of tracks being promoted or evicted. Again we note that promotions and evictions are interruptible background operations; the impact on host I/O will be a loss of throughput due to these operations, plus RMW latency for writes to non-promoted tracks.

2.4 Dynamic track mapping

Dynamic track mapping is another strategy for addressing the track flipping key limitations: (1) only neighboring tracks could be switched and (2) a small portion of bottom track must remain unflipped. It achieves this by allowing arbitrary permutations of tracks within *zones* (groups of small numbers of tracks).

To address unequal track sizes, in dynamic mapping we concatenate all bottom-track LBAs and group them in fixed-sized pseudo-tracks of approximately one physical track in size (except for the last pseudo-track). We similarly group all top-track LBAs into pseudo-tracks of the same size. These equal-sized pseudo-tracks may then be arbitrarily switched with each other. Algorithm 3 describes dynamic track mapping in more detail; as seen, we periodically check for hot bottom pseudo-tracks and swap the hottest bottom with the coldest top tracks.

Memory usage: If zones are sized to hold 256 pseudo-tracks, only 8 bits are needed for each map entry; for our 20 TB drive with almost 13M tracks this would require about 12.5 MB of memory: more than that needed for track flipping, but still modest.

Safety and crash consistency: Similar to track flipping and track caching, to persist the changes, dynamic track mapping logs the updates to the map and also writes a full checkpoint periodically.

Timing: The time required to swap a hot bottom track T with a cold top track T' in dynamic track mapping is very similar to that of track flipping. However, since pseudo-track size is not equal to physical track size, it is possible that a track cannot be read immediately after the head is placed resulting in a half rotation on average. The time required for a single swap will be:

1. $t_{seek} + 0.5t_{rot}$ to reach track T
2. $3t_{rot}$ to read tracks $T-1$, T, and $T+1$
3. t_{seek} to reach a backup region

4. $3t_{rot}$ to write backup copies of track $T-1$, T and $T+1$
5. $t_{seek}+0.5t_{rot}$ to reach track T'
6. t_{rot} to read track T'
7. $t_{seek}+0.5t_{rot}$ to return to track T
8. $3t_{rot}$ to write the contents of T' into track T and write back the contents of $T-1$ and $T+1$
9. $t_{seek}+0.5t_{rot}$ to return to track T'
10. t_{rot} to write the contents of T into track T'

Thus, the expected total cost is $5t_{seek}$+$13t_{rot}$.

3 Methodology

We evaluate the four IMR translation algorithms—naive read-modify-write (RMW), track flipping, selective track caching and dynamic track mapping— in addition to conventional disk (CMR) via trace-driven simulation. We use the CloudPhysics traces [18], a recent set of block traces from virtual machines running Linux and Windows with modern file systems and large storage volumes. The LBA ranges covered in the traces varied from tens of gigabytes to 1.5TB.

Prior work [6] has shown that older traces (e.g. the widely-used MSR Cambridge traces [13] from c. 2007) display fine-grained behavior which is very different from that exhibited by modern file systems; although this difference in behavior may not be significant in block-level systems (e.g. FTLs) that ignore spatial locality, it has been demonstrated

Algorithm 3: dynamic track mapping

```
parameter :updateFrequency, swapThreshod
variable  :ioDirection (read/write), trackPosition,
           trackNumber, writeCount, trackIdLog
           [], trackCounts [], cldstTrk, hotstTrk
1  ioDirection, trackPosition, trackNumber ← ReceiveIO()
2  Function RemapTracks() is
3  |  trackCounts [] ← Count (trackIdLog)
4  |  for every cldstTrk in Coldest (trackCounts)
   |    and hotstTrk in Hottest (trackCounts) do
5  |  |  temperatureGap = hotstTrkCntr- cldstTrkCntr
6  |  |  if temperatureGap > swapThreshod then
7  |  |  |  TrackSwap (cldstTrk, hotstTrk)
8  |  |  end
9  |  end
10 end
11 if ioDirection == write then
12 |  writeCount ++
13 |  trackIdLog.append(trackNumber)
14 |  if writeCount mod updateFrequency == 0 then
15 |  |  for every zone do
16 |  |  |  RemapTracks()
17 |  |  end
18 |  end
19 end
```

to result in significant differences in the performance of disk-based systems. The CloudPhysics corpus comprises 106 different traces; we sampled this set and selected a collection that represents different levels of read/write intensity and spatial locality. A summary of the selected workloads is shown in Table 1. The workloads range in size from about 3 to 44 million I/Os, and range from read-heavy (w08, 09% writes) to very write-heavy (w39, 95% writes).

Disk model: Our simulation assumes a 6000 RPM disk ($T_{rot} = 10$ms) with equal-sized 2 MB tracks; although crude, we argue that this model is fairly accurate in the absence of real IMR disks for comparison. Based on current trends, 2 MB is a reasonable estimate of the mean track size for a next-generation drive; however real disks have decreasing track sizes towards the inner radius of the platter, with roughly a factor of two difference between the largest and smallest tracks. Since the majority of sectors lie in the larger outer tracks, the actual variance from the mean is less than this factor of two would imply, and as modern file systems (ext4 and NTFS) do not consider track location (as opposed to locality) in placement, errors in either direction are expected to cancel out.

The primary inaccuracy introduced by this model is in track flipping: the model assumes that top and bottom tracks are of equal size, so that no remainder of the bottom track is left behind after flipping. If top tracks in a real drive have 90% the capacity of bottom tracks, this would result in up to 10% of writes to this track being classified by the simulator as top-track writes, rather than bottom-track RMW writes. Since the colder end of the track is left behind, we expect that the misclassification rate be less than 10%. However, if both track ends are equally hot, the misclassification rate will be higher. Our observation of hot-track LBA access patterns suggests that either one end is extremely hot or all LBAs are accessed evenly. At present we neither know the actual ratio of top to bottom track size in a specific real IMR drive, nor the practical range of this parameter for feasible drives; therefore this 90% figure is highly speculative. Given this uncertainty, the choice of a uniform track size is simple and not unreasonable.

Trace playback: The traces used were collected in a virtualized environment, with a high-performance multi-disk (or SSD) back-end storage system. The resulting I/O rates may be seen in the inter-arrival time CDFs in Figure 6, where half of inter-arrival times for one trace (w84) are in the 100 μS range (up to 10,000 IOPS), while 80% of writes for another trace (w35) are below 500 μS (2000 IOPS). Several approaches may be taken in adapting such a trace to a single-disk simulation. One method is to run the simulation "flat-out", ignoring inter-arrival times and launching (or queuing) each I/O as soon as possible. However since our IMR translation algorithms include background work, the resulting behavior would not be representative of system behavior for real applications.

Our goal instead is to simulate what system behavior would be if the application that produced the original trace were run against the simulated (and much slower) I/O device.

Table 1: Statistical summary of selected workloads.

workload	w08	w09	w17	w24	w26	w28	w31	w34	w39	w43	w46	w48	w56	w61	w84	w87	w106
I/O count (M)	44.3	49.6	31.3	27.1	26.5	19.7	21.1	19.5	17.9	15.6	11.5	14	10.8	9.8	4.8	3.7	3.2
write ratio	0.09	0.55	0.78	0.11	0.58	0.33	0.16	0.23	0.97	0.51	0.62	0.42	0.95	0.50	0.86	0.79	0.82

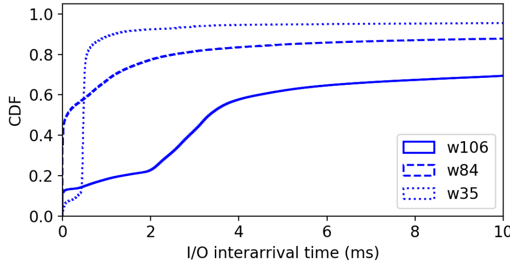


Figure 6: CDF of I/O inter-interval times in a few workloads.

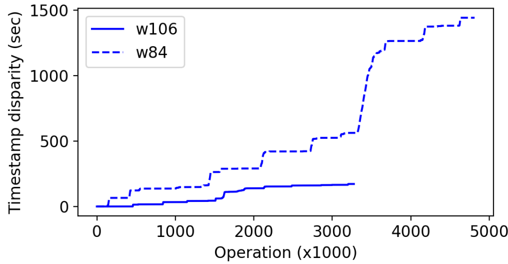


Figure 7: "Stretching" of original trace timestamps for IMR simulation.

Accurate simulation of such behavior requires information on application CPU usage and I/O dependencies which are not available in the original traces. Although recent work [5] provides mechanisms for inferring I/O dependencies given a trace containing both initiation and completion times for I/Os, very few of the traces we used contained this information.

Instead we consider a fixed-queue model, with a queue size of 64; based on inspection of the traces we believe this is an intermediate point between the lower effective queue depths seen with many application-initiated I/Os, and the very high degree of parallelism with which I/Os are initiated by the virtual memory system. I/O inter-arrival times are preserved when inserting items into the queue until it is filled, and then further I/Os are blocked until a position in the queue opens; in other words, the inter-arrival time between I/Os N and N+1 is the maximum of the original trace inter-arrival time and the time for the queue to drain by 1. The resulting I/O performance reflects a combination of individual I/Os and queuing delays due to device throughput limitations. As an example, in Figure 7 we see this "time dilation" for two workloads with naive RMW—in one case trace completion is delayed by about 150s, and in the other by over 1440s.

3.1 Disk model details

I/O latency: In our simulation, I/O latency includes host and device queuing, seek time, rotational delay and transfer time. Seek time (T_{seek}) is calculated from the source and destination track locations using the following equation proposed by Shafaei [16], assuming minimum and maximum seek times of 2 and 20 ms and calculating α accordingly:

$$t_{seek}(trk_{src}, trk_{des}) = \alpha * \sqrt{|trk_{src} - trk_{des}|} + t_{seek_{min}} \tag{1}$$

Rotational delay for I/Os on different tracks is assumed to be uniformly distributed between 0 and 1 rotation ($T_{rotation}$); for deterministic simulation we assume a constant value of a half rotation, giving a total I/O latency of:

$$t_{I/O} = t_{seek} + \frac{1}{2} t_{rotation} + t_{transfer} \tag{2}$$

We note that IOs are split at boundaries in case they touch more than a track. The full list of drive specifications and experiment configurations is shown in Table 2.

4 Evaluation

In this section we evaluate five alternatives—conventional disk (CMR), IMR with naive read-modify-write (RMW), IMR with track flipping, IMR with selective track caching and IMR with dynamic track mapping—by measuring I/O latencies and write amplification factor (WAF). Summary results for all traces and algorithms are shown in Figure 8 (write amplification) and Figure 9 (latency).

All track flipping, selective track caching and dynamic track mapping are seen to give substantial improvements in write

Table 2: Experimental parameters and drive specification.

drive specification		
track size	drive cache size	rotation delay
2MB	100MB	10 ms
dynamic track mapping and flipping configuration		
update frequency	hot/cold threshold	max flips
20K write ops	50	50
selective track caching configuration		
update frequency	track cache size	cache location
20K write ops	100 tracks	OD

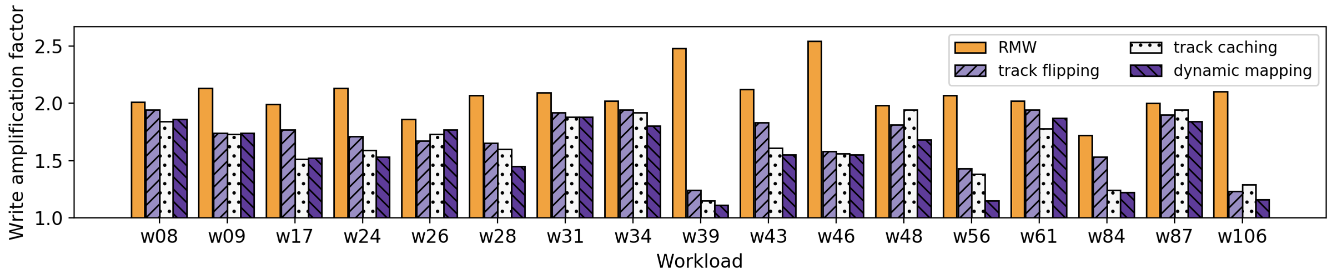
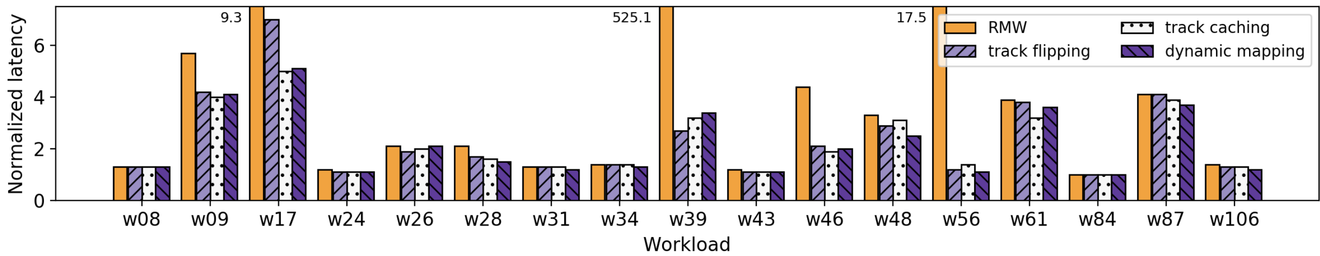
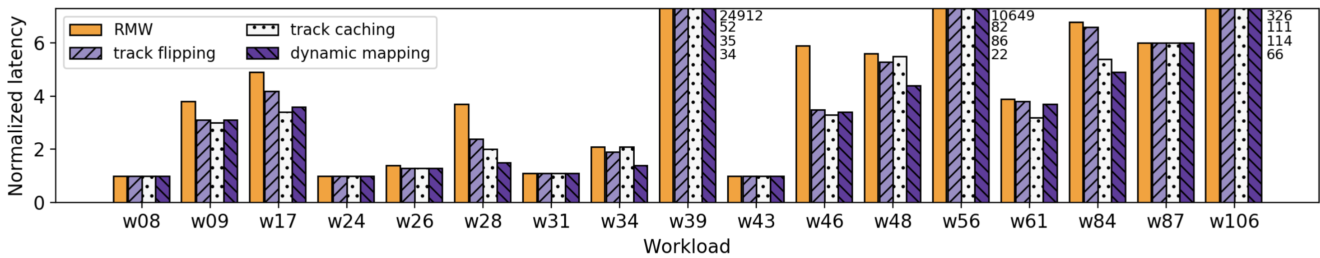


Figure 8: IMR write amplification by workload and translation layer.



(a) Mean latency



(b) 90th percentile latency

Figure 9: Mean and tail latency for 17 workloads: CMR, RMW, track flipping, track caching and dynamic track mapping.

amplification when compared to naive read-modify-write, by a factor of 2 or more in over half of the cases; in no case is performance degraded. As expected, dynamic track mapping shows the best performance in most cases as it has the minimum limitations among the proposed approaches. For several workloads it reduces the write amplification by a factor of 2 or more; in some cases (e.g., w56, w39 and w106) write amplification is nearly eliminated.

Results for mean latency are more mixed. In several cases (e.g., w09, w17, w39, w46, w56, w61 and w87) IMR read-modify-write latencies were noticeably higher than for the conventional drive. For w56 this excess latency was virtually eliminated by track flipping, track caching or dynamic track mapping. For some others (e.g., w17, w39, w46) at least one of the approaches gave a significant reduction (more than 2x) in IMR latency, while still remaining about twice that of conventional. For w09 and w87, however, latency improvements from dynamic track mapping were modest, with performance still significantly worse than CMR.

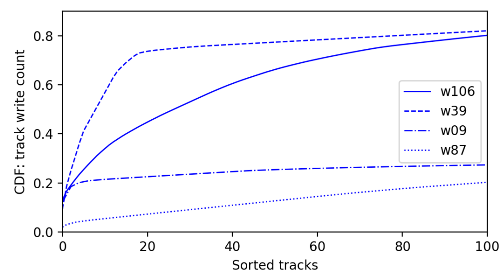


Figure 10: CDF of track write counts for workloads with highest (w106 and w39) and lowest (w87 and w87) WAF improvement with track flipping and/or selective track caching.

In Figure 10 we explore one possible reason for lower improvement for traces w09 and w87: the overall number of hot tracks. We compare these traces with w106 and w39, the traces that show the greatest improvements in write amplification when track flipping or track caching is used (see

Figure 8). We see that a high fraction of writes in w106 are to the hottest 100 tracks, and that half of the writes in w39 are to a handful of tracks. In contrast the "working set" of hot tracks for w09 and w87 appears to be very large, with only around 20% and 15% of writes going to the hottest 100 tracks, respectively. The track cache used in our experiments (100 tracks) would do almost nothing to help in this case, and if many writes are to tracks which are written only a few times at most, then the gain from flipping them will not outweigh the cost.

As with mean latency, results for tail latency (Figure 9b) are also mixed, although worse. In a few cases (e.g., w39 and w56) RMW increases tail latency by a factor of tens of thousands. That is because with CMR most writes in these cases appear to complete in the write cache which results in a mean latency of about 0.1ms, and there is just enough idle time for them to be flushed to disk; however with RMW the disk cannot keep up during idle time, so the queue fills and stays that way causing mean latency of about 2.4s. Overall tail latency is increased by more than 4x across majority of the workloads. In the worst cases (w39, w56) the proposed algorithms limit the relative increase in tail latency to double digits, but is still very high. In about half the cases tail latency with dynamic track mapping is similar to that of CMR; however in the remainder it is significantly worse.

For both mean and tail latency, there are a few workloads (w08, w24, w31 and w43) that are not affected by IMR and accordingly no improvement is observed when the proposed algorithms are applied. Besides w43 with almost 51% of write operations, the rest of the workloads are read-heavy traces and therefore are less prone to significant performance penalty due to IMR.

To examine further, in Figure 12 we see CDFs of I/O latencies for traces w46 and w28. In each case IMR read-modify-write results in high-latency I/Os due to a combination of operation latency and queuing delays due to reduced throughput; roughly 2/3 of writes were slowed in both cases. We note that there are some other cases e.g., w56 with fewer writes (roughly 10%) being affected. Track flipping, track caching and dynamic track mapping are all able to improve the w28 and w46 performance considerably, but a significant fraction of writes (roughly 25%) still suffer excessive latency. Our observations show that for the case w56 the three improved algorithms are all able to eliminate the excess latencies, resulting in performance comparable to a conventional drive.

The impact of IMR overhead on throughput can be approximated by looking at the issue time expansion during the simulation run; this indicates the periods at which the device was unable to keep up with the I/O trace, and by how much. In Figure 13 we see issue time disparity vs. I/O count for traces w46 and w28. We note that for w46 all dynamic track mapping, and to a slightly lesser extent track caching and track flipping, result in significantly improved throughput. For w28, dynamic track mapping shows a considerable throughput improvement; track caching and track flipping show a smaller improvement. We also note that throughput of the simulated algorithms would increase with larger I/Os, as read-modify-writes would be amortized over larger I/O sizes.

Track flipping only works if hot tracks are adjacent to cold tracks; if hot regions on the disk are substantially larger than a track, this might not be the case. In Figure 4 of Section 2.2 we saw that this was the case for trace 17; however in Figure 11 we see the same analysis for w87; although the hottest few tracks stand alone, many tracks with very high write counts are surrounded by tracks of similar hotness.

5 Related work

Interlaced Magnetic Recording [9] is a new storage technology using HAMR (Heat assisted magnetic recording) [4, 12] and track overlap (the technique on which SMR [19] is based) to achieve higher areal density than possible with either approach alone [3]. Numerous works have characterized [1] and modeled [15, 16] SMR performance; however due to fundamental differences in track layout and write constraints such work is not directly applicable to IMR.

While only a limited number of translation layers and data management techniques have been proposed for IMR in the two years since the original work became public [20], a wide range of file systems and translation layers have been proposed for SMR, such as Cassuto's indirection system [2], SMaRT [8] from He and Du, Shafaei's Virtual Guard [14], and FSTL [7]. Cassuto et al. propose a set associative persistent cache to hold updated sectors. SMaRT [8] proposes using a track-based dynamic mapping. Shafaei et al. propose a track-based static mapping translation layer which caches tracks containing at-risk data, rather than the track targeted by the I/O. Hajkazemi et al. [7] propose an LBA-based translation layer based on dynamic mapping. Since the write restrictions in SMR are a strict superset of IMR restrictions, SMR translation layers could in fact be applied to IMR; however this would ignore the performance improvements possible due to lessened write restrictions.

To the best of our knowledge, the data management design introduced by Wu et al. [20] is the only published work on IMR translation layers to date. The authors propose *Top-Buffer*, a technique utilizing unallocated top tracks of each track-group (a small set of tracks interlaced with top tracks) as a buffer to store LBA updates corresponding to bottom tracks. Moreover they suggest *Block Swapping*, a technique to swap bottom hot LBAs with cold ones within a track-group. Our work differs in that it is targeted for in-disk implementation, in a restricted-memory environment, while the memory requirements for Wu et al.'s algorithm are beyond the capabilities of a drive controller.

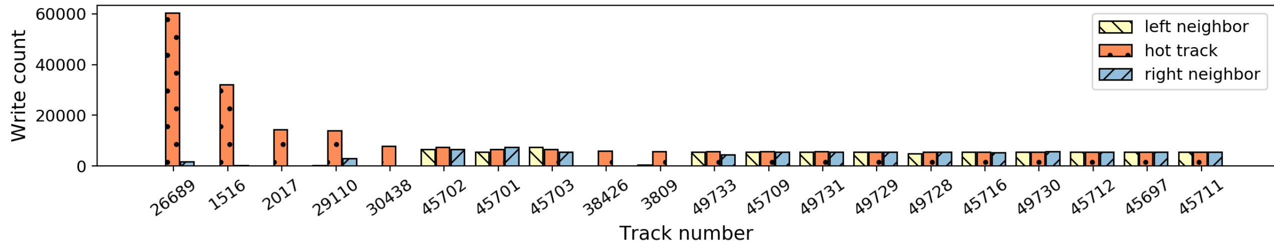
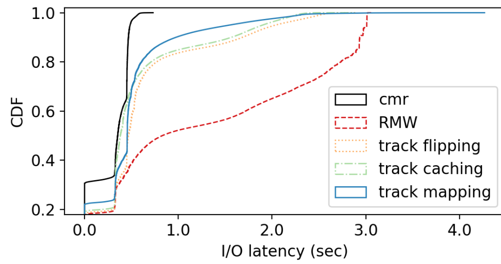
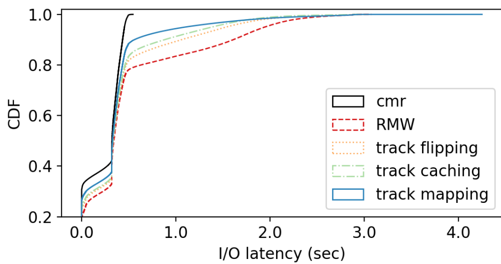


Figure 11: Write count of 20 hottest tracks and their neighbors, trace 87. This trace is seen to be "track flipping-unfriendly".

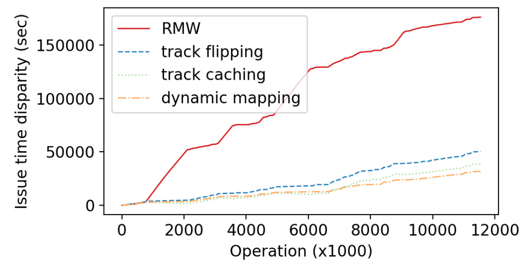


(a) w46

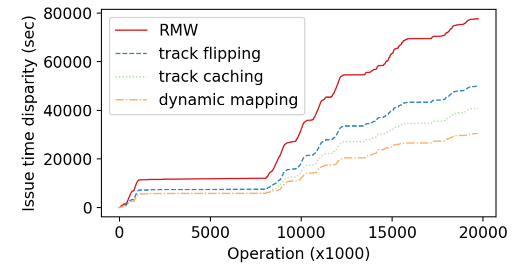


(b) w28

Figure 12: I/O latency distribution: workloads w46 (a) and w28 (b), for CMR, IMR/RMW, IMR/track flipping, IMR/track caching.



(a) w46



(b) w28

Figure 13: Issue time disparity (I/O issue time gap between CMR and other studied approaches) of traces w46 and w28.

6 Conclusion

Interlaced magnetic recording is still a new—or even speculative—technology, and we cannot be sure of its precise characteristics until real prototypes are available. However, when such prototypes arrive, algorithms will be needed to cope with the IMR write restrictions, and due to the track-based nature of the restrictions, those algorithms will need to run in the memory-limited environment of the drive controller.

We quantify the performance of the naive read-modify-write algorithm for IMR bottom track writes, showing that it is significantly more costly than assumed in prior work, and show via trace-driven simulations that for some workloads its performance is comparable to that of a conventional disk, but that it is worse, sometimes catastrophically so, for others. We present three algorithms to reduce the frequency of IMR bottom-track writes: *track flipping*, *selective track caching* and *dynamic track mapping*, with sufficiently modest memory requirements to be readily implemented in drive controllers. These algorithms are shown to improve I/O amplification significantly for almost all workloads examined, and to improve latency for some—but not all—of the workloads which performed poorly with IMR read-modify-write. Further research is needed to determine whether extensions of this work (e.g. track flipping+caching) will yield conventional drive-level performance for IMR with acceptable memory cost.

Acknowledgment

We would like to thank Irfan Ahmad and CloudPhysics for the use of their traces, our shepherd William Jannen, and the anonymous reviewers for their valuable suggestions.

References

[1] AGHAYEV, A., SHAFAEI, M., AND DESNOYERS, P. Skylight—a window on shingled disk operation. *ACM Transactions on Storage (TOS) 11*, 4 (2015), 16.

[2] CASSUTO, Y., SANVIDO, M. A. A., GUYOT, C., HALL, D. R., AND BANDIC, Z. Z. Indirection systems for shingled-recording disk drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–14.

[3] GRANZ, S., JURY, J., REA, C., JU, G., THIELE, J., RAUSCH, T., AND GAGE, E. C. Areal density comparison between conventional, shingled, and interlaced heat-assisted magnetic recording with multiple sensor magnetic recording. *IEEE Transactions on Magnetics 55*, 3 (March 2019), 1–3.

[4] GRANZ, S., ZHU, W., SENG, E. C. S., KAN, U. H., REA, C., JU, G., THIELE, J.-U., RAUSCH, T., AND GAGE, E. C. Heat-assisted interlaced magnetic recording. *IEEE Transactions on Magnetics 54*, 2 (2018), 1–4.

[5] HAGHDOOST, A., HE, W., FREDIN, J., AND DU, D. H. C. On the Accuracy and Scalability of Intensive I/O Workload Replay. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 315–328.

[6] HAJKAZEMI, M. H., ABDI, M., AND DESNOYERS, P. Minimizing read seeks for smr drives. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization* (2018), IEEE.

[7] HAJKAZEMI, M. H., ABDI, M., SHAFAEI, M., AND DESNOYERS, P. Fstl: A framework to design and explore shingled magnetic recording translation layers. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)* (Oct. 2018), IEEE.

[8] HE, W., AND DU, D. H. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, 2017), USENIX Association, pp. 121–134.

[9] HWANG, E., PARK, J., RAUSCHMAYER, R., AND WILSON, B. Interlaced magnetic recording. *IEEE Transactions on Magnetics 53*, 4 (2017), 1–7.

[10] JACOB, B., NG, S., AND WANG, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[11] KREVAT, E., TUCEK, J., AND GANGER, G. R. Disks Are Like Snowflakes: No Two Are Alike. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2011), HotOS XIII, USENIX Association, pp. 14–14.

[12] KRYDER, M. H., GAGE, E. C., MCDANIEL, T. W., CHALLENER, W. A., ROTTMAYER, R. E., JU, G., HSIA, Y.-T., AND ERDEN, M. F. Heat assisted magnetic recording. *Proceedings of the IEEE 96*, 11 (2008), 1810–1835.

[13] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (San Jose, California, 2008), USENIX Association, pp. 1–15.

[14] SHAFAEI, M., AND DESNOYERS, P. Virtual Guard: A Track-Based Translation Layer for Shingled Disks. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017), USENIX Association.

[15] SHAFAEI, M., HAJKAZEMI, M. H., DESNOYERS, P., AND AGHAYEV, A. Modeling smr drive performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (New York, NY, USA, 2016), SIGMETRICS '16, ACM, pp. 389–390.

[16] SHAFAEI, M., HAJKAZEMI, M. H., DESNOYERS, P., AND AGHAYEV, A. Modeling drive-managed smr performance. *ACM Transactions on Storage (TOS) 13*, 4 (2017), 38.

[17] THOMPSON, D., AND BEST, J. The future of magnetic data storage techology. *IBM Journal of Research and Development 44*, 3 (May 2000), 311–322.

[18] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 95–110.

[19] WOOD, R., WILLIAMS, M., KAVCIC, A., AND MILES, J. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics 45*, 2 (2009), 917–923.

[20] WU, F., ZHANG, B., CAO, Z., WEN, H., LI, B., DIEHL, J., WANG, G., AND DU, D. H. C. Data Management Design for Interlaced Magnetic Recording. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, Feb. 2018), USENIX Association.

Your Coflow Has Many Flows: Sampling Them for Fun and Speed

Akshay Jajoo
ajajoo@purdue.edu

Y. Charlie Hu
ychu@purdue.edu
Purdue University

Xiaojun Lin
linx@purdue.edu

Abstract

Coflow scheduling improves data-intensive application performance by improving their networking performance. State-of-the-art online coflow schedulers in essence approximate the classic Shortest-Job-First (SJF) scheduling by learning the coflow *size* online. In particular, they use multiple priority queues to simultaneously accomplish two goals: to sieve long coflows from short coflows, and to schedule short coflows with high priorities. Such a mechanism pays high overhead in learning the coflow size: moving a large coflow across the queues delays small and other large coflows, and moving similar-sized coflows across the queues results in inadvertent round-robin scheduling.

We propose PHILAE, a new online coflow scheduler that exploits the spatial dimension of coflows, *i.e.,* a coflow has many flows, to drastically reduce the overhead of coflow size *learning*. PHILAE pre-schedules sampled flows of each coflow and uses their sizes to estimate the average flow size of the coflow. It then resorts to Shortest Coflow First, where the notion of shortest is determined using the learned coflow sizes and coflow contention. We show that the sampling-based learning is robust to flow size skew and has the added benefit of much improved scalability from reduced coordinator-local agent interactions. Our evaluation using an Azure testbed, a publicly available production cluster trace from Facebook shows that compared to the prior art Aalo, PHILAE reduces the coflow completion time (CCT) in average (P90) cases by 1.50× (8.00×) on a 150-node testbed and 2.72× (9.78×) on a 900-node testbed. Evaluation using additional traces further demonstrates PHILAE's robustness to flow size skew.

1 Introduction

1.1 Motivation

In big data analytics jobs, speeding up the communication stage where the data is transferred between compute nodes is important to speed up the jobs. However, improving network level metrics such as flow completion time may not translate into improvements at the application level metrics such as job completion time. The coflow abstraction [18] was proposed to bridge such a gap. The abstraction captures the collective network requirements of applications, as reduced coflow completion time (CCT) can directly lead to faster job completion time [20, 24].

There have been a number of efforts on network designs for coflows [7, 21, 27] that assume complete prior knowledge of coflow sizes (The coflow size is defined as the total size of its constituent flows.). However, in many practical settings, coflow characteristics are not known a priori. For example, multi-stage jobs pipeline data from one stage to the next as soon as the data is generated, which makes it difficult to know the size of each flow [22, 40]. A recent study [40] shows various other reasons why it is not very plausible to learn flow sizes from applications, for example, learning flow sizes from applications requires changing either the network stack or the applications.

Scheduling coflows in such *non-clairvoyant* settings, however, is challenging. The major challenge in developing an effective non-clairvoyant coflow scheduling scheme has centered around how to learn the coflow sizes online quickly and accurately, as once the coflow sizes (bytes to be transferred) can be estimated, one can apply variations of the classic Shortest-Job-First (SJF) algorithm such as Shortest Coflow First [21] or apply an LP solver (*e.g.,* [7]).

State-of-the-art online non-clairvoyant schedulers such as Saath [30], Gravtion [29] and Aalo [19] in essence learn coflow sizes and approximate SJF using discrete priority queues, where all newly arriving coflows start from the highest priority queue, and move to lower priority queue as they send more data (without finishing), *i.e.,* cross the per-queue thresholds. In this way, the smaller coflows finish in high priority queues, while the larger coflows gradually move to the lower priority queues where they finish after smaller coflows.

To realize the above idea in scheduling coflows which have flows at many network ports, *i.e.,* in a distributed setting, Aalo uses a global coordinator to assign coflows to logical priority queues, and uses the total bytes sent by all flows of a coflow as its logical "length" in moving coflows across the queues. The logical priority queues are mapped to local priority queues at each port, and the individual local ports then schedule the flows in its local priority queues, *e.g.,* by enumerating flows from the highest to lowest priority queues and using FIFO to order the flows within each queue.

In essence, Aalo learns coflow sizes by actually scheduling the coflow, a "try and miss" approach to approximate SJF. As coflow sizes are not known, in each queue, Aalo schedules each coflow for a fixed amount of data (try). If the coflow does not finish (miss), it is demoted to a lower priority queue.

Afterwards, such a coflow will no longer block coflows in higher priority queues.

Using multiple priority queues to learn the relative coflow sizes of coflows this way, however, negatively affects the average CCT and the scalability of the coordinator:

(1) Intrinsic queue-transit overhead: Every coflow that Aalo transits through the queues before reaching its final queue worsens the average CCT because during transitions, such a coflow effectively *blocks other shorter* coflows in the earlier queues it went through, which would have been scheduled before this coflow starts in a perfect SJF.

(2) Overhead due to inadvertent round-robin: Although Aalo attempts to approximate SJF, it inadvertently ends up doing *round-robin* for coflows of similar sizes as it moves them across queues. Aalo assigns a *fixed threshold of data transfer* for each coflow in each queue. Assume there are "N" coflows in a queue that do not finish in that queue. Aalo schedules one coflow (chosen using FIFO) and demotes it to a lower priority queue when the coflow reaches the data threshold. At that point, the next coflow from the same queue is scheduled, which joins the previous coflow at a lower priority queue after exhausting its quantum, and this cycle continues as coflows of similar sizes move through the queues. Effectively, these coflows experience the round-robin scheduling which is known to have the worst average CCT [39], when jobs are of similar sizes.

(3) Limited scalability from frequent updates from local ports: To support the try-and-error style learning, the coordinator requires frequent updates from all local ports of the bytes sent for each coflow in order to move coflows across multiple queues timely. This results in high load on the central coordinator from receiving frequent updates and calculating and sending new rate allocations, which limits the scalability of the overall approach.

Empirical measurement We quantify the coflow size *learning overhead* of Aalo, defined as the portion of the bytes of a coflow that has been transferred (or the fraction of its CCT spent in doing so) before reaching its correct queue, using a trace from Facebook clusters [4] (see detailed methodology in §8). Figure 1 shows that 40% of the coflows that moved beyond the initial queue reached the correct priority queue after spending more than 20% of their CCT moving across early queues.

1.2 Our Contribution

We propose Philae, a new non-clairvoyant coflow scheduler with a dramatically different approach to learning coflow sizes to enable online SJF. To leverage optimal scheduling SJF in coflow scheduling, it is vital to learn the coflow sizes quickly and accurately. Philae achieves this objective by exploiting the *spatial dimension* of coflows, *i.e.,* a coflow typically consists of many flows, via *sampling*, a highly effective technique used in large-scale surveys [34]. In particular, Philae pre-schedules sampled flows, called *pilot flows*, of each coflow and uses their measured size to estimate the coflow size. It then resorts to SCF using the estimated job size.

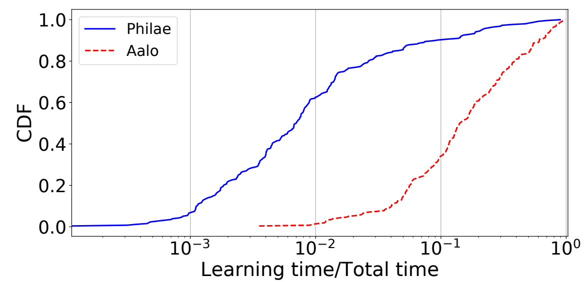


Figure 1: CDF of learning overhead per coflow, *i.e.,* the time to reach the correct priority queue as a fraction of CCT, excluding coflows directly scheduled by Philae or finish in Aalo's first queue.

Intuitively, such a sampling scheme avoids all three sources of overhead in Aalo – Once the coflow sizes are learned, the coflows are assigned to the correct queues, which avoids the intrinsic queue-transit and round-robin effects. Further, a sampling-based design has an important benefit – it offers much higher scalability than priority-queue-based learning in Aalo. This is because unlike Aalo, after estimating the coflow size, Philae clients do not need to send periodic updates of bytes sent-so-far to the centralized coordinator.

Developing a complete non-clairvoyant coflow scheduler based on the simple sampling idea raises three questions: (1) Why is sampling more efficient than the priority-queue-based coflow size learning? (2) Will sampling be effective in the presence of skew of flow sizes? (3) How to design the complete scheduler architecture? We systematically address these questions with design rational, theoretical analysis, system design, prototyping, and extensive evaluation.

In summary, this paper makes the following contributions:

(1) Using a production datacenter trace from Facebook, we show that the prior art scheduler Aalo spends substantial amount of time and network bandwidth in learning coflow sizes, which negatively affects the CCT of coflows.
(2) We propose the novel idea of applying sampling in the spatial dimension of coflow to significantly reduce the overhead of online learning coflow sizes.
(3) We present theoretical underpinning explaining why sampling remains effective in the presence of flow size skew.
(4) We present the design and implementation of Philae.
(5) We extensively evaluate Philae via simulations and testbed experiments, and show that compared to the prior art, the new design reduces the average CCT by 1.51× for the Facebook coflow trace and by 1.36× for a trace with properties similar to a Microsoft production cluster.
(6) The CCT improvement mainly stems from reduced coflow size learning overhead. Philae reduces the median latency and data sent in finding the right queue for coflows in Aalo by 19.0× and 20.0×, respectively (§8.2).

2 Background and Problem Statement

We start with a brief review of the coflow abstraction and the need for non-clairvoyant coflow scheduling. We then state the network model and problem formulation.

Coflow abstraction In data-parallel applications such as Hadoop [1] and Spark [2], the job completion time heavily depends on the completion time of the communication stage [12, 20]. The coflow abstraction [18] was proposed to speed up the communication stage to improve application performance. A coflow is defined as a set of flows between several nodes that accomplish a common task. For example, in map-reduce jobs, the set of all flows from all map to all reduce tasks in a single job forms a typical coflow. The coflow completion time (CCT) is defined as the time duration between when the first flow arrives and the last flow completes. In such applications, improving CCT is more important than improving individual flows' completion time (FCT) for improving the application performance [19, 21, 24, 29, 30].

Non-clairvoyant coflows Data-parallel directed acyclic graphs (DAGs) typically have multiple stages which are represented as multiple coflows with dependencies between them. Recent systems (*e.g.,* [3, 22, 28, 36]) employ optimizations that pipeline the consecutive computation stages which removes the barrier at the end of each coflow, making knowing flow sizes of each coflow beforehand difficult. Thus in this paper, we focus on *non-clairvoyant* coflow scheduling which do not assume knowledge about coflow characteristics such as flow sizes upon coflow arrival.

Non-blocking network fabric We assume the same non-blocking network fabric model in recent network designs for coflows [7, 19, 21, 29, 30], where the datacenter network fabric is abstracted as a single non-blocking switch that interconnects all the servers, and each server (computing node) is abstracted as a network port that sends and receives flows. In such a model, the ports, *i.e.,* server uplinks and downlinks, are the only source of contention as the network core is assumed to be able to sustain all traffic injected into the network. We note that the abstraction is to simplify our description and analysis, and is not required or enforced in our evaluation.

Problem statement Our goal is to *develop an efficient non-clairvoyant coflow scheduler that optimizes the communication performance, in particular the average CCT, of data-intensive applications without prior knowledge, while guaranteeing starvation freedom and work conservation and being resilient to the network dynamics.* The problem of non-clairvoyant coflow scheduling is NP-hard because coflow scheduling even assuming all coflows arrive at time 0 and their size are known in advance is already NP-hard [21]. Thus practical non-clairvoyant coflow schedulers are approximation algorithms. Our approach is to dynamically prioritize coflows by efficiently learning their flow sizes online.

3 Key Idea

Our new non-clairvoyant coflow scheduler design, Philae, is based on a key observation about coflows that a coflow has a *spatial dimension, i.e.,* it typically consists of many flows. We thus propose to explicitly learn coflow sizes online by using *sampling*, a highly effective technique used in large-scale surveys [34]. In particular, Philae preschedules sampled flows, called *pilot flows*, of each coflow and uses their measured sizes to estimate the coflow size. It then resorts to SJF or variations using the estimated coflow sizes.

Developing a complete non-clairvoyant coflow scheduler based on the simple sampling idea raises three questions:

(1) Why is sampling more efficient than the priority-queue-based coflow size learning? Would scheduling the remaining flows after sampled pilot flows are completed adversely affect the coflow completion time?

(2) Will sampling be effective in the presence of skew of flow sizes?

(3) How to design the complete scheduler architecture?

We answer the first two questions below, and present the complete architecture design in §4.

3.1 Why is sampling more efficient?

Scheduling pilot flows first before the rest of the flows can potentially incur two sources of overhead. First, scheduling pilot flows of a newly arriving coflow consumes port bandwidth which can delay other coflows (with already estimated sizes). However, compared to the multi-queue based approach, the overhead is much smaller for two reasons: (1) Philae schedules only a small subset of the flows (*e.g.,* fewer than 1% for coflows with many flows). (2) Since the CCT of a coflow depends on the completion of its last flow, some of its earlier finishing flows could be delayed without affecting the CCT. Philae exploits this observation and schedules pilot flows on the least-busy ports to increase the odds that it only affects earlier finishing flows of other coflows.

Second, scheduling pilot flows first may elongate the CCT of the newly arriving coflow itself whose other flows cannot start until the pilot flows finish. This is again typically insignificant for two reasons: (1) A coflow (*e.g.,* from a MapReduce job) typically consists of flows from all sending ports to all receiving ports. Conceptually, pre-scheduling one out of multiple flows from each sender may not delay the coflow progress at that port, because all flows at that port have to be sent anyway. (2) Coflow scheduling is of high relevance in a busy cluster (when there is a backlog of coflows in the network), in which case the CCT of coflow is expected to be much higher than if it were the only coflow in the network, and hence the piloting overhead is further dwarfed by a coflow's actual CCT.

3.2 Why is sampling effective in the presence of skew?

The flow sizes within a coflow may vary (*skew*). Intuitively, if the skew across flow sizes is small, sampling even a small number of pilot flows will be sufficient to yield an accurate estimate. Interestingly, even if the skew across flow sizes is large, our experiment indicates that sampling is still highly effective. In the following, we give both the intuition and theoretical underpinning for why sampling is effective.

Consider, for example, two coflows and the simple setting where both coflows share the same set of ports. In order to improve the average CCT, we wish to schedule the shorter coflow ahead of the longer coflow. If the total sizes of the two coflows are very different, then even a moderate amount of estimation error of the coflow sizes will not alter their ordering. On the other hand, if the total sizes of the two coflows are close to each other, then indeed the estimation errors will likely alter their ordering. However, in this case since their sizes are not very different anyway, switching the order of these two coflows will not significantly affect the average CCT.

Analytic results. To illustrate the above effect, we show that the gap between the CCT based on sampling and assuming perfect knowledge is small, even under general flow size distributions. Specifically, coflows C_1 and C_2 have cn_1 and cn_2 flows, respectively. Here, we assume that n_1 and n_2 are fixed constants. Thus, by taking c to be larger, we will be able to consider wider coflows. Assume that each flow of C_1 (correspondingly, C_2) has a size that is distributed within a bounded interval $[a_1, b_1]$ ($[a_2, b_2]$) with mean μ_1 (μ_2), *i.i.d.* across flows. However, the exact distributions can be arbitrary. Let T^c be the total completion time when the exact flow sizes are known in advance. Let $\tilde{T}^c$ be the average CCT by sampling m_1 and m_2 flows from C_1 and C_2, respectively. Without loss of generality, we assume that $n_2\mu_2 \geq n_1\mu_1$. Then, using Hoeffding's Inequality, we can show that,

$$\lim_{c\to\infty} \frac{\tilde{T}^c - T^c}{T^c} \leq 4\exp\left[-\frac{2(n_2\mu_2 - n_1\mu_1)^2}{\left(\frac{n_2(b_2-a_2)}{\sqrt{m_2}} + \frac{n_1(b_1-a_1)}{\sqrt{m_1}}\right)^2}\right] \frac{n_2\mu_2 - n_1\mu_1}{n_2\mu_2 + 2n_1\mu_1} \tag{1}$$

(Note that here we have used the fact that, since both coflows share the same set of ports and c is large, the CCT is asymptotically proportional to the coflow size.)

Equation (1) can be interpreted as follows. First, due to the first exponential term, the relative gap between $\tilde{T}^c$ and T^c decreases as $b_1 - a_1$ and $b_2 - a_2$ decrease. In other words, as the skew of each coflow decreases, sampling becomes more effective. Second, when $b_1 - a_1$ and $b_2 - a_2$ are fixed, if $n_2\mu_2 - n_1\mu_1$ is large (i.e., the two coflow sizes are very different), the value of the exponential function will be small. On the other hand, if $n_2\mu_2 - n_1\mu_1$ is close to zero (i.e., the two coflow sizes are close to each other), the numerator on the second term on the right hand side will be small. In both cases, the relative gap between $\tilde{T}^c$ and T^c will also be small, which is consistent with the intuition explained earlier. The largest gap occurs when $n_2\mu_2 - n_1\mu_1$ is on the same order as $\frac{n_2(b_2-a_2)}{\sqrt{m_2}} + \frac{n_1(b_1-a_1)}{\sqrt{m_1}}$. Finally, although these analytical results assume that both coflows share the same set of ports, similar conclusions on the impact of estimation errors due to sampling also apply under more general settings.

The above analytical results suggest that, when c is large, the relative performance gap for CCT is a function of the number of pilot flows sampled for each coflow, but is independent of the total number of flows in each coflow. In practice, large coflows will dominate the total CCT in the system. Thus, these results partly explain that, while in our experiments the number of pilot flows is never larger than 1% of the total number of flows, the performance of our proposed approach is already very good.

Finally, the above analytical results do not directly tell us how to choose the number of pilot flows, which likely depends on the probability distribution of the flow size. In practice, we do not know such distribution ahead of time. Further, while choosing a larger number of pilot flows reduces the estimation errors, it also incurs higher overhead and delay. Therefore, our design (§4) needs to have practical solutions that carefully address these issues.

4 Philae Design

In this section, we present the detailed design of Philae, which addresses three design challenges: (1) *Coflow size estimation:* How to choose and schedule the pilot flows for each newly arriving coflow? (2) *Starvation avoidance:* How to schedule coflows after size estimation using variations of SJF that avoid starvation? (3) *Coflow scheduling:* How to schedule among all the coflows with estimated sizes?

4.1 Philae architecture

Fig. 2 shows the Philae architecture. Philae models the entire datacenter as a single big-switch with each computing node as an individual port. The scheduling task in Philae is divided among (1) a central coordinator, and (2) local agents that run on individual ports. A computing framework such as Spark [42] first registers (removes) a coflow when a job arrives (finishes). Upon a new coflow arrival, old coflow completion, or pilot flow completion, the coordinator calculates a new coflow schedule, which includes (1) coflows that are to be scheduled in the next time slot, and (2) flow rates for the individual flows of a coflow, and pushes this information to the local agents which use this information to allocate their bandwidth. The local agents will follow the current schedule until they receive a new schedule.

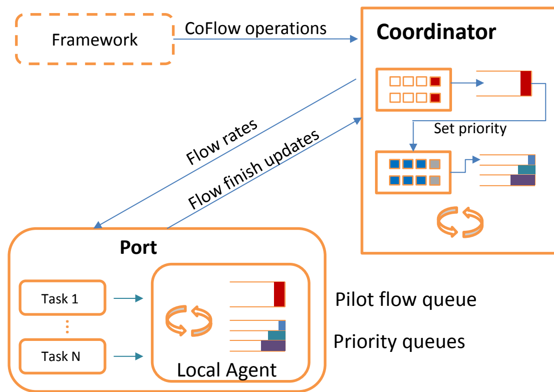


Figure 2: PHILAE architecture.

4.2 Sampling pilot flows

As discussed in §3, PHILAE estimates the size of a coflow online by actually scheduling a subset of its flows (*pilot flows*) at their ports. We do not schedule the flows of a coflow other than the pilot flows until the completion of the pilot flows in order to avoid unnecessary extra blocking of other potentially shorter coflows.

How many pilot flows? When a new coflow arrives, PHILAE first needs to determine the number of pilot flows. As discussed at the end of §3, the number of pilot flows affects the trade-off between the coflow size estimation accuracy and scheduling overhead. For coflows with skewed flow sizes, accurately estimating the total coflow size potentially requires sampling the sizes of many pilot flows. However, scheduling pilot flows has associated overhead, *i.e.,* if the coflow turns out to be a large coflow and should have been scheduled to run later under SJF.

We explore several design options for choosing the number of pilot flow. Two natural design choices are using a constant number of pilot flows or a fixed fraction of the total number of flows of a coflow. In addition, we observe that typical coflows consist of flows between a set of senders (*e.g.,* mappers) and a set of receivers (*e.g.,* reducers) [23]. We thus include a third design choice of a fixed fraction of sending ports. This design also spreads the pilot flows to avoid having multiple pilot flows contending for the same sending ports. We empirically found that (§8.2) limiting the pilot flows to 5% to 10% of the number of its sending ports (*e.g.,* mappers in a MapReduce coflow) strikes a good balance between estimation accuracy and overhead. We note the total number of flows sampled in this case is still under 1%.

Finally, we *estimate the total coflow size as* $S = f_i \cdot N$, where N is the number of flows in a coflow, and f_i is the average size of the sampled pilot flows.

Which flows to probe? Second, PHILAE needs to decide which ports to schedule the chosen number of probe flows for a coflow. For this, we use a simple heuristic where, upon the arrival of a new coflow, we select the ports for its pilot flows that are least busy, *i.e.,* having pilot flows from the least number of other coflows. PHILAE starts with the least busy sending port and iterates over receiving ports starting with the least busy receiving port and assigns the flow if it exists. It then updates the statistics for the number of pilot flows scheduled at each port and repeats the above process. Such a choice will likely delay fewer coflows when the pilot flows are scheduled and hence reduce the elongation on their CCT. We note that such an online heuristic may not be optimal; more sophisticated algorithms can be derived by picking ports for multiple coflows together. However, we make this design choice for its simplicity and low time complexity to ensure that the coordinator makes fast decisions.

How to schedule pilot flows? In PHILAE, we prioritize the pilot flows of a new coflow over existing flows to accelerate learning the size of the new coflow. In particular, at each port, pilot flows have high priority over non-pilot flows. If there are multiple outstanding pilot flows (of different coflows) at a port, PHILAE schedules them in the FIFO order.

4.3 Coflow scheduling with starvation avoidance

Once the sizes of coflows are learned, we can apply variations of the SJF policy to schedule them. However, it is well known that such policies can lead to starvation.

There are many ways to mitigate the starvation issue. However, a subtlety arises where *even slight difference in how starvation is addressed can result in different performance.* For example, the multiple priority queues in Aalo has the benefit of ensuring progress of all coflows, but assigning different time-quanta to different priority queues can result in different average CCT for the same workload. To ensure the fairness of performance comparison with Aalo, we need to ensure that both PHILAE and Aalo provide the same level of starvation freedom (or progress measure).

For this reason, in this paper, we inherit the multiple priority queue structure from Aalo for coflow scheduling. As in Aalo, PHILAE sorts the coflows among multiple priority queues. In particular, PHILAE uses N queues, Q_0 to Q_{N-1}, with each queue having lower queue threshold Q_q^{lo} and higher threshold Q_q^{hi}, where $Q_0^{lo} = 0$, $Q_{N-1}^{hi} = \infty$, $Q_{q+1}^{lo} = Q_q^{hi}$, and the queue thresholds grow exponentially, *i.e.,* $Q_{q+1}^{hi} = E \cdot Q_q^{hi}$.

The overall coflow scheduling in PHILAE works as follows. After the coflow size is estimated using pilot flows, PHILAE assigns the coflow to the priority queue using inter-coflow policies discussed in §4.4. Within a queue, we use FIFO to schedule coflows. Lastly, we use weighted sharing of network bandwidth among the queues, where a priority queue receives a network bandwidth based on its priority. As in Aalo, the weights decrease exponentially with decrease in the priority of the queues.

Using FIFO within the priority queue and weighted fair sharing among the queues together ensure the same starva-

tion freedom and thus meaningful performance comparison between PHILAE and Aalo [19].

4.4 Inter-coflow scheduling policies

In PHILAE, we explore four different scheduling policies based on different combinations of coflow size and contention, two size-based policies (*A, B*) as in Aalo, a contention-based, similar to the intra-queue policy used in Saath [30] (*C*), and a new contention-and-length-based policy (*D*):

(A) Smallest job first: Coflows are sorted based on coflow size ($l \cdot n$).

(B) Smallest remaining data first: Coflows are sorted based on remaining data ($l \cdot n - d$).

(C) Least contention first: Coflows are sorted based on their contention (c).

(D) Least length-weighted total-port contention first: Coflows are sorted based on the sum of port-wise contention times estimated flow length $\sum_p c^p \cdot l$.

We use the following parameters of a coflow to define the metrics in scheduling algorithms: (1) average flow length (l) from piloting, (2) number of flows (n), (3) number of sender and receiver ports (s,r), (4) total amount of data sent so far (d), (5) contention (c), defined as the number of other coflows sharing any ports with the given coflow, and (6) port-wise contention (c^p), defined as the number of other coflows blocked at a given port p.

PHILAE uses Policy D by default, as it results in the least average CCT (§8). For all policies, we continue to use the priority-queue based scheduling, and the algorithms only differ in what metric they use in assigning coflows to the priority queues. In contrast, Aalo does not handle inter-coflow contention, and uses the total bytes sent so far (d) to move coflows across multiple priority queues.

4.5 Rate allocation

Once the scheduling order of the coflows is determined, we need to determine the rates for the individual flows at each port. First, since we want to quickly finish the pilot flow, at any port that has pilot flows, PHILAE assigns the *entire* port bandwidth to the pilot flows. For the remaining ports, as discussed in §4.3, across multiple queues, PHILAE assigns weighted shares of the port bandwidth, by assigning them varying numbers of scheduling intervals according to the weights assigned to each priority queues.

Second, at each scheduling interval, PHILAE assigns rates for the flows of the coflow in the head of the FIFO queue as follows. It assigns equal rates at all the ports containing its flows as there is no benefit in speeding-up its flows at certain ports when its CCT depends on the slowest flow. At each port, we could use max-min fairness to schedule the individual flows of the coflow (to different receivers), and then assign the rate of the slowest flow to all the flows in the coflow. Afterwards, the port-allocated bandwidths are incremented accordingly at the coordinator, which then allocates rates for the next coflow in the same FIFO queue, and so on.

Though the above max-min approach has the advantage of minimizing bandwidth wastage, it slows down the coordinator which has to iterate over many flows. In our experiments, we used a simple scheme where we assign the *entire* bandwidth at the sender and receiver ports to one flow of the coflow at the head of the FIFO queue at a time. We found that this simple scheme has very marginal effect on CCTs but makes the rate assignment process considerably faster.

4.6 Additional design issues

Thin coflow bypass Recall that, in PHILAE, when a new coflow arrives, PHILAE only schedules its pilot flows. All other flows of that coflow are delayed until the pilot flows finish and coflow size is known. However, such a design choice can inadvertently lead to higher CCTs for coflows, particularly for thin coflows, *e.g.,* a two-flow coflow would end up serializing scheduling its two flows, one for the piloting purpose.

To avoid CCT degradations for thin coflows, we schedule all flows of a coflow if its width is under a threshold (set to 7 in PHILAE; §8.6 provides sensitivity analysis for thresholds).

Failure tolerance and recovery Cluster dynamics such as stragglers or node failure can delay some of the flows of a coflow or start new flows, increasing their CCT. The PHILAE design automatically self-adjusts to speed up coflows that are affected by cluster dynamics using the following mechanisms: (1) It adjusts the coflow size as the amount of data left by the coflow, which is essentially the difference between the size calculated using pilot flows and amount of data already sent. (2) It calculates contention only on the ports that have unfinished flows.

Work Conservation By default, PHILAE schedules non-pilot flows of a coflow only after all its pilot flows are over. This can lead to some ports being idle where the non-pilot flows are waiting for the pilot flows to finish. In such cases, PHILAE schedules non-pilot flows of coflows which are still in the sampling phase at those ports. In work conservation, the coflows are scheduled in the FIFO order of arrival of coflows.

5 Scalability Analysis

Compared to learning coflow sizes using priority queues (PQ-based) [19, 30], learning coflow sizes by sampling PHILAE not only reduces the learning overhead as discussed in §3.1 and shown in §8.2, but also significantly reduces the amount of interactions between the coordinator and local agents and thus makes the coordinator highly scalable, as summarized in Table 1.

First, *PQ-based learning requires much more frequent update from local agents.* PQ-based learning estimates coflow sizes by incrementally moving coflows across priority queues

Table 1: Comparison of frequency of interactions between the coordinator and local agents.

	Update of data sent	Update of flow completion	Rate calculation
Philae	No	Yes	Event triggered
Aalo	Periodic (δ)	Yes	Periodic (δ)

according to the data sent by them so far. As such, the scheduler needs frequent updates (every δ ms) of data sent per coflow from the local agents. In contrast, Philae directly estimates a coflow's size upon the completion of all its pilot flows. The only updates Philae needs from the local agents are about the flow completion which is needed for updating contentions and removing flows from active consideration..

Second, *PQ-based learning results in much more frequent rate allocation.* In sampling-based approach, since coflow sizes are estimated only once, coflows are re-ordered only upon coflow completion or arrival events or in the case of contention based policies only when contention changes, which is triggered by completion of all the flows of a coflow at a port. In contrast, in PQ-based learning, at every δ interval, coflow data sent are updated and coflow priority may get updated, which will trigger new rate assignment.

Our scalability experiments in §9.3 confirms that Philae achieves much higher scalability than Aalo.

6 Implementation

We implemented both Philae and Aalo scheduling policies in the same framework consisting of the global coordinator and local agents (Fig. 2), in 5.2 KLoC in C++.

Coordinator: The coordinator schedules the coflows based on the operations received from the registering framework. The key implementation challenge for the coordinator is that it needs to be fast in computing and updating the schedules. The Philae coordinator is optimized for speed using a variety of techniques including pipelining, process affinity, and concurrency whenever possible.

Local agents: The local agents update the global coordinator only upon completion of a flow, along with its length if it is a pilot flow. Local agents schedule the coflows based on the last schedule received from the coordinator. They comply to the last schedule until a new schedule is received. To intercept the packets from the flows, local agents require the compute framework to replace `datasend()`, `datarecv()` APIs with the corresponding Philae APIs, which incurs very small overhead.

Coflow operations: The global coordinator runs independently from, and is not coupled to, any compute framework, which makes it general enough to be used with any framework. It provides RESTful APIs to the frameworks for coflow operations: (a) `register()` for registering a new coflow when it enters, (b) `deregister()` for removing a coflow when it exits, and (c) `update()` for updating coflow status whenever there is a change in the coflow structure, *e.g.*, during task migration and restarts after node failures.

7 Evaluation Highlights

We evaluated Philae using a 150-node and a 900-node testbed cluster in Azure and using large scale simulations by utilizing a publicly available Hive/MapReduce trace collected from a 3000-machine, 150-rack Facebook production cluster [4] and multiple derived traces with varying degrees of flow size skew to measure Philae's robustness to skew.

- **Facebook (FB) trace**: The trace contains 150 ports and 526 ($> 7 \times 10^5$ flows) coflows, that are extracted from Hive/MapReduce jobs from a Facebook production cluster. Each coflow consists of pair-wise flows between a set of senders and a set of receivers.

Due to the lack of other publicly available coflow trace[1], we derived three additional traces using the original Facebook trace in order to more thoroughly evaluate Philae under varying coflow size skew:

- **Low-skew-filtered**: Starting with the FB trace, we filtered out coflows that have skew (*max flow length*/*min flow length*) less than a constant k. We generated five traces in this class with $k = 1,2,3,4,5$. The filtered traces have 142, 100, 65, 51 and 43 coflows, respectively.

- **Mantri-like**: Starting with the FB trace, we adjusted the sizes of the flows sent by the mappers, keeping the total reducer data the same as given in the original trace, to match the skew of a large Microsoft production cluster trace as described in Mantri [12]. In particular, the sizes are adjusted so that the coefficients of variation across mapper data are about 0.34 in the 50th percentile case and 3.1 in the 90th percentile case. This trace has the same numbers of coflows and ports as the FB trace.

- **Wide-coflows-only**: We filtered out all the coflows in the FB trace with the total number of flows ≤ 7, the default thin coflow bypass threshold (thinLimit) in Philae. The filtered trace has 269 coflows spreading over 150 ports.

The primary performance metrics used in the evaluation are CCT or CCT speedup, defined as the ratio of a CCT under other baseline algorithms and under Philae, piloting overhead, and coflow size estimation accuracy.

The highlights of our evaluation results are:

(1) Philae significantly improves the CCTs. In simulation using the FB trace, the average CCT is improved by 1.51× over the prior art, Aalo. Individual CCT speedups are 1.78× in the median case (P90 = 9.58×). For the Mantri-like trace,

[1] A challenge that has also been faced by previous work on coflow scheduling such as [19, 27, 29, 44].

Table 2: Performance improvement over Aalo for varying pilot flow selection schemes.

	Constant	Proportional to number of senders					Proportional to number of flows	
	2	5%	10%	20%	50%	100%	1%	10%
Avg. error	13.21%	6.14%	5.42%	4.94%	5.53%	4.25%	4.15%	2.90%
Avg. CCT	1.27x	1.51x	1.45x	1.50x	1.50x	1.50x	1.43x	0.49x
P50 speedup	1.75x	1.78x	1.76x	1.71x	1.52x	1.40x	1.33x	0.69x
P90 speedup	9.00x	9.58x	9.00x	9.15x	8.33x	8.45x	8.23x	8.23x

the average CCT is improved by 1.36× and individual CCT speedups are 1.75× in the median case (P90 = 12.0×).

(2) The CCT improvement mainly stems from the reduction in the learning overhead (in terms of latency and amount of data sent) in determining the right queue for the coflows. Compared to Aalo, median reduction in the absolute latency in finding the right queue for coflows in PHILAE is 19.0×, and in absolute amount of data sent is 20.0× (§8.2).

(3) PHILAE improvements are consistent when varying the skew among the flow sizes in a coflow (§8.5).

(4) PHILAE improvements are consistent when varying its parameters (§8.6).

(5) The PHILAE coordinator is much more scalable than that of Aalo (§9.3).

8 Simulation

We present detailed simulation results in this section, and the testbed evaluation of our prototype in §9.

Experimental setup: Our simulated cluster uses the same number of nodes (sending and receiving network ports) as in the trace. As in [19], we assume full bisection bandwidth is available, and congestion can happen only at network ports.

The *default parameters* for Aalo and PHILAE in the experiments are: starting queue threshold (Q_0^{hi}) is 10MB, exponential threshold growth factor (E) is 10, number of queues (K) is set to 10, the weights assigned to individual priority queues decrease exponentially by a factor of 10, and the new schedule calculation interval δ is set to 8ms for the 150-node cluster [2], the default suggested in its publicly available simulator [19]. In PHILAE, a new schedule is calculated on demand, upon arrival of a new coflow, completion of a coflow, or completion of all pilot flows of a coflow. Finally, in PHILAE the threshold for thinLimit (T) is set to 7, the number of pilot flows assigned to wide coflows are $max(1, 0.05 \cdot S)$, where S is the number of senders, and the default inter-coflow scheduling policy in PHILAE is Least length-weighted total-port contention.

8.1 Pilot flow selection policies

We start by evaluating the impact of different policies in choosing the pilot flows for a coflow in PHILAE. Table 2 summarizes the improvement in average CCT of PHILAE over Aalo and average error in size estimation normalized to the actual coflow size, when varying the pilot flow selection policy while keeping other parameters as the default in PHILAE, using the FB trace.

Unsurprisingly, the estimation accuracy increases when increasing the number of pilot flows across the three selection schemes: constant, fraction of senders, and fraction of total flows. However, as the number of pilot flows increases (over the range of parameter choices), the CCT speedup (P50 and P90 of individual coflow CCT speedups) decreases. This is because the benefit from size estimation accuracy improvement from using additional pilot flows does not offset the added overhead from completing the additional pilot flows and the delay they incur to other coflows.

We find sampling 5% of the number of senders per coflow strikes a good trade-off between piloting overhead and size estimation accuracy leading to the best CCT reduction. We thus set it ($0.05 \cdot S$) as the default pilot flow selection policy.

8.2 Piloting overhead and accuracy

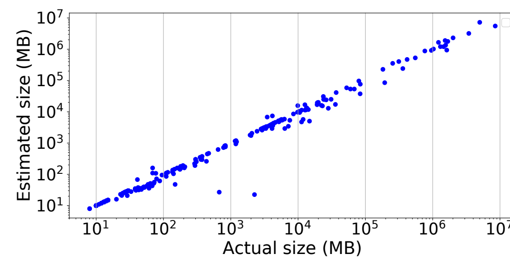


Figure 3: PHILAE coflow size learning accuracy. Coflows that did not go through the piloting phase (48%) are not shown.

Next, using the default pilot selection policy, we evaluate PHILAE's effectiveness in estimating coflow sizes by sampling pilot flows. Fig. 3 shows a scatter plot of the actual coflow size vs. estimated size from running PHILAE under the default settings. We observe that PHILAE coflow's size estimation is highly accurate except for a few outliers. Overall, the average and standard deviation of relative estimation error are 0.06 and 0.15, respectively, and for the top 99% and 95% coflows (in terms of estimation accuracy), the average (standard deviation) of relative error are only 0.05 (0.12) and 0.03 (0.07) respectively. Interestingly, a few coflows experience large estimation errors, and we found they all have very high skew

[2]8ms is the time to send 1MB of data.

in their flow lengths; the mean standard deviation in flow lengths, normalized by the average length, of the bottom 1% (in terms of accuracy) ranges between 4.6 and 6.8.

Fig. 1 shows the cost of estimating the correct queue for each coflow in Philae and Aalo, measured as the time in learning the coflow size as a fraction of the coflow's CCT in Philae and Aalo. We see that under Philae, about 63% of the coflows spent less than 1% of their CCT in the learning phase, while under Aalo, 63% coflows reached the correct priority queue after spending up to 22% of their CCT moving across other queues. Compared to Aalo, Philae in the median case sends 20× less data in determining the right queue and reduces the latency in determining the right queue by 19×.

8.3 Inter-coflow scheduling policies

Philae differs from Aalo in two ways: online size estimation and inter-flow scheduling policy. Here, we evaluate the effectiveness of the four inter-coflow scheduling policies of Philae discussed in §4.4, keeping the remaining parameters as the default. Such evaluation allows us to decouple the contribution of sampling-based learning from the effect of scheduling policy difference.

Table 3 shows the CCT improvement of Philae under the four inter-flow scheduling policies over Aalo. We make the following observations.

First, Philae with the purely sized-based policy, **Smallest job first (A)**, which uses the same inter-queue and intra-queue scheduling policy as Aalo and only differs from Aalo in coflow size estimation, reduces the average CCT (P50) of Aalo by 1.40x (1.48x).

In contrast, the default Philae uses **Least length-weighted total-port contention (D)**, which uses the sum of size-weighted port contention to assign coflows to priority queues, and slightly outperforms the size-based policy A; it reduces the average CCT (P50) of Aalo by 1.51x (1.78x). This is because it captures the diversity of contention at different ports, which happens often in real distributed settings, and at the same time accounts for the coflow size by using length-weighted sum of the port-wise contention. The above results for policy A and policy D indicate that the primary improvement in Philae comes from its sampling-based coflow size estimation scheme.

Shortest remaining time first (B) performs similarly as smallest job first. This is because the preemptive nature of SRTF will kick in only on arrival of new coflows. Also, although SRTF is advantageous for small coflows, since Philae already schedules thin coflows at high priority, many thin and thus small coflows are anyways being scheduled at high priority under both policies A and B, and as a result they perform similarly.

Finally, **Least contention first (C)** performs poorly. This is because contention for a coflow is defined as the unique number of other coflows that share ports, and as a result such a policy completely ignores the size (length) of the coflows.

Table 3: CCT speedup in Philae under different inter-coflow scheduling policies (§4.4) over Aalo.

Priority estimation metric	P50	P90	Avg. CCT
Estimated size (A)	1.48x	8.27x	1.40x
Remaining size (B)	1.54x	8.34x	1.37x
Global Contention (C)	0.75x	8.26x	0.13x
Length-weighted total-port contention (D) (Philae)	1.78x	9.58x	1.51x

8.4 Average CCT improvement

We now compare the CCT speedups of Philae against 5 well-known coflow scheduling policies: (1) Aalo, (2) Aalo-Oracle, which is an oracle version of Aalo where the scheduler knows the final queue of a coflow upon its arrival time and directly starts the coflow from that queue, (3) SEBF in Varys [21] which assumes the knowledge of coflow sizes apriori and uses the Shortest Effective Bottleneck First policy, where the coflow whose slowest flow will finish first is scheduled first. (4) FIFO, which is a single queue FIFO based coflow scheduler, and (5) FAIR, which uses per-flow fair sharing. We do not include Saath [30] in the comparison as it does not provide the same liveliness guarantees as Philae which as discussed in §4.3 can obscure the comparison result. All experiments use the default parameters discussed in the setup, including K, E, S, unless otherwise stated. The results are shown in Fig. 4(a). We make the following observations.

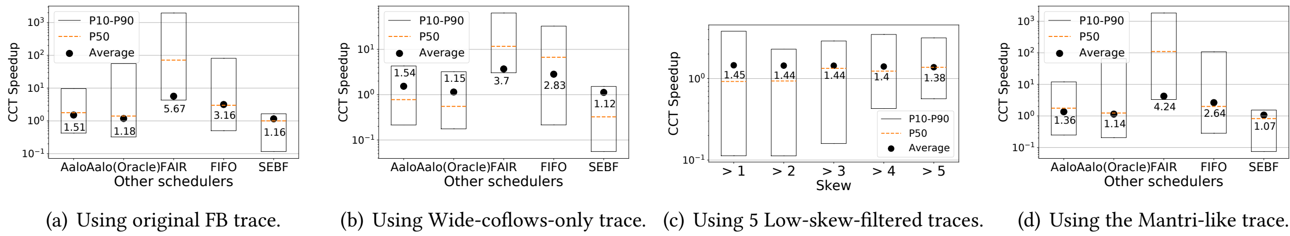
First, we compare CCT under Philae against under Aalo-Oracle, where Aalo-Oracle starts all coflows at the correct priority queues (*i.e.,* no learning overhead). Philae improves the average CCT by 1.18× and P50 CCT by 1.40×, respectively. Since Aalo-Oracle pays no overhead for coflow size estimation, its worse performance suggests that using length-weighted total-port contention in assigning coflows to the priority queues in Philae outperforms Aalo's size-based, contention-oblivious policy in assigning coflows to the queues.

Second, Philae improves the average CCT over Aalo by 1.51× (median) and P50 by 1.78. The significant *additional* improvement on top of the gain over Aalo-Oracle comes from fast and accurate estimation of the right queues for the coflows (Fig. 1).

Third, Philae, which requires no coflow size knowledge a priori, achieves comparable performance as SEBF [21]; it reduces the average CCT by 1.16×. Again this is because its total-port contention policy outperforms the contention-oblivious SEBF.

Finally, Philae significantly outperforms the single-queue FIFO-based coflow scheduler, with a median (P90) CCT speedup of 3.00 (77.96)× and average CCT speedup of 3.16×, and the un-coordinated flow-level fair-share scheduler, with a median (P90) CCT speedup of 70.82× (1947×) and average CCT speedup of 5.66×.

To gain insight into how different coflows are affected by Philae over Aalo, we group the coflows in the trace into



(a) Using original FB trace. (b) Using Wide-coflows-only trace. (c) Using 5 Low-skew-filtered traces. (d) Using the Mantri-like trace.

Figure 4: CCT speedup using Philae compared to using other coflow schedulers on different traces. In Fig. 4(c), the x-axis denotes the minimum skew in the 5 Low-skew-filtered traces.

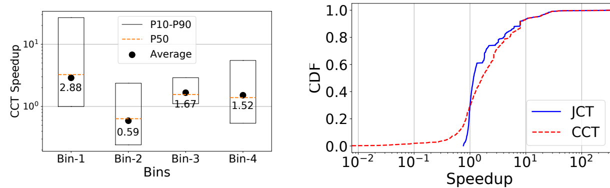


Figure 5: Performance breakdown into bins shown in Table 4.

Figure 6: [Testbed] Distribution of speedup in CCT and JCT in Philae using the FB trace.

Table 4: Bins based on total coflow size and width (number of flows). The numbers in brackets denote the fraction of coflows in that bin.

	width $\leq$ 7(thin)	width $>$ 7(wide)
size $\leq$ 100MB (small)	bin-1 (44.3%)	bin-2 (24.1%)
size $>$ 100MB (large)	bin-3 (4.5%)	bin-4 (27.1%)

four bins defined in Table 4, and show in Fig. 5 the CCT speedups for each bin. We see that Philae improves CCT for all coflows in bin 1 and 3 and for large fraction in bin-4. Most of the underperforming coflows fall in bin-2. Coflows in bin-2 have width $>$ 7 and size $<$ 100MB, *i.e.,* the flows are short but wide. Because the width exceeds the thinLimit, Philae schedules the pilot flows to estimate the coflow size first (§4). Thus, although the remaining flows are short, they get delayed until the completion of the pilot flows, which results in CCT increase.

Finally, since thin coflows benefit from Philae's scheme of bypassing probing for thin coflows, we also compare Philae with other schemes using the Wide-coflows-only trace which consists of all coflows wider than the default thinLimit (7) in Philae. Fig. 4(b) shows that Philae continues to perform well, reducing the average CCT by 1.54×, 1.15×, and 1.12× over Aalo, Aalo-Oracle, and SEBF, respectively.

8.5 Robustness to coflow data skew

Next, we evaluate Philae's robustness to flow size skew by comparing it against Aalo using traces with varying degrees of skew. First, we evaluate Philae using the Mantri-like trace. Fig. 4(d) shows that Philae consistently outperforms prior-art coflow schedulers. In particular, Philae reduces the average CCT by 1.36x compared to Aalo. Second, we evaluate Philae using the Low-skew-filtered traces which have low skew coflows filtered out. Fig. 4(c) shows that Philae performs better than Aalo even with highly skewed traces and reduces the average CCT by 1.45×, 1.44×, 1.44×, 1.40× and 1.38× for the five Low-skew-filtered traces containing coflows with skew of at least 1, 2, 3, 4 and 5, respectively.

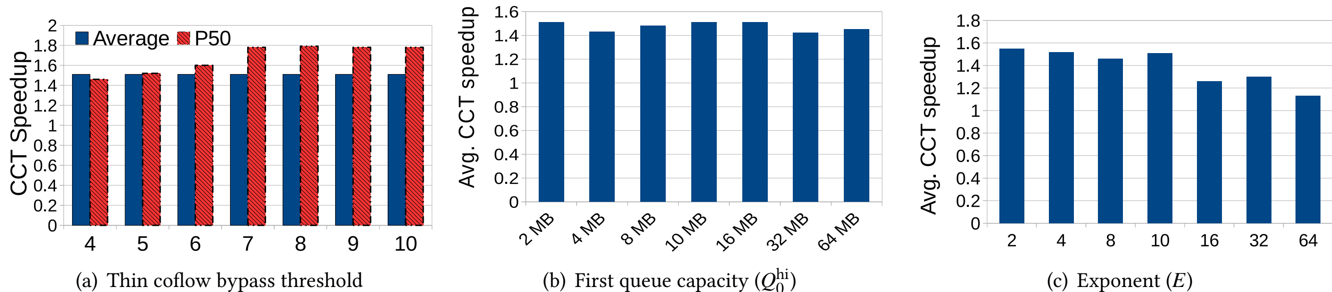
8.6 Sensitivity analysis

Compared to Aalo, Philae has only two additional paramaters: thinLimit and flow sampling rate. We already discussed the choice of sampling rate in §8.1. Below, we evaluate the sensitivity of Philae to thinLimit and other design parameters common to Aalo by varying one parameter at a time while keeping the rest as the default.

Thin coflow bypassing limit (T) In this experiment, we vary thinLimit (T) in Philae for bypassing coflows from the probing phase. The result in Fig. 7(a) shows that the average CCT remains almost the same as T increases. This is because the average CCT is dominated by wide and large coflows, which are not affected by thinLimit. However, the P50 speedup increases till $T = 7$ and tapers off after $T = 7$. The reason for the CCT improvement until $T = 7$ is that all flows of thin coflows (with width ≤ 7) are scheduled immediately upon arrival which improves their CCT, and the number of thin coflows is significant.

Start queue threshold (Q_0^{hi}) We next vary the threshold for the first priority queue from 2 MB to 64 MB. Fig. 7(b) shows the average CCT of Philae over Aalo. Overall, Philae is not very sensitive to the threshold of first priority queue and the CCT speedup over Aalo is within 8% of the default Philae (10 MB). The speedup appears to oscillate with a periodicity of 5x to 10x. For example, the speedups for 2 MB and 64 MB are close to that of the default (10 MB), while for 4 MB and 32 MB are lower. This can be explained by the impact of the first queue threshold on job segregation; with the default queue threshold growth factor of 10, every time the first queue threshold changes by close to 10x, the distribution of jobs across the queues become similar.

Multiplication factor (E) In this experiment, we vary the queue threshold growth factor from 2 to 64. Recall that the queue thresholds are computed as $Q_q^{hi} = Q_{q-1}^{hi} \cdot E$. Thus, as E grows, the number of queues decreases. As shown in Fig. 7(c),



(a) Thin coflow bypass threshold (b) First queue capacity (Q_0^{hi}) (c) Exponent (E)

Figure 7: [Simulation] PHILAE sensitivity analysis. We vary one parameter of PHILAE keeping rest same as default and compare it with Aalo.

smaller queue threshold multiplication factor which leads to more queues performs better because of fine-grained priority segregation.

9 Testbed Evaluation

Next, we deployed PHILAE in a 150-machine Azure cluster and a 900-machine cluster to evaluate its performance and scalability.

Testbed setup: We rerun the FB trace on a Spark-like framework on a 150-node cluster in Microsoft Azure [5]. The coordinator runs on a Standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory. The local agents run on D2v2 with the same processor as the coordinator with 2-core and 7GB memory. The machines on which local agents run have 1 Gbps network bandwidth. Similarly as in simulations, our testbed evaluation keeps the same flow lengths and flow ports in trace replay. All the experiments use default parameters K, E, S and the default pilot flow selection policy.

9.1 CCT Improvement

In this experiment, we measure CCT improvements of PHILAE compared to Aalo. Fig. 6 shows the CDF of the CCT speedup of individual coflows under PHILAE compared to under Aalo. The average CCT improvement is 1.50× which is similar to the results in the simulation experiments. We also observe 1.63× P50 speedup and 8.00× P90 speedup.

We also evaluated PHILAE using the Wide-coflow-only trace. Table 5 shows that PHILAE achieves 1.52× improvement in average CCT over Aalo, similar to that using the full FB trace. This is because the improvement in average CCT is dominated by large coflows, PHILAE is speeding up large coflows, and the Wide-coflow-only trace consists of mostly large coflows.

9.2 Job Completion Time

Next, we evaluate how the improvement in CCT affects the job completion time (JCT). In data clusters, different jobs spend different fractions of their total job time in data shuffle. In this experiment, we used 526 jobs, each corresponding to one coflow in the FB trace. The fraction of time that the jobs spent in the shuffle phase follows the same distribution used in Aalo [19], *i.e.,* 61% jobs spent less than 25% of their total time in shuffle, 13% jobs spent 25-49%, another 14% jobs spent 50-74%, and the remaining spent over 75% of their total time in shuffle. Fig. 6 shows the CDF of individual speedups in JCT. Across all jobs, PHILAE reduces the job completion time by 1.16× in the median case and 7.87× in the 90th percentile. This shows that improved CCT translates into better job completion time. As expected, the improvement in job completion time is smaller than the improvement in CCT because job completion time depends on the time spent in both compute and shuffle (communication) stages, and PHILAE improves only the communication stage.

Table 5: [Testbed] CCT improvement in PHILAE as compared to Aalo.

	P50	P90	Avg. CCT
FB Trace	1.63×	8.00×	1.50×
Wide-coflow-only	1.05×	2.14×	1.49×

Table 6: [Testbed] Average (standard deviation) coordinator CPU time (ms) per scheduling interval in 900-port runs. PHILAE did not have to calculate and send new rates in 66% of intervals, which contributes to its low average.

	Rate Calc.	New Rate Send	Update Recv.	Total
PHILAE	2.99 (5.35)	4.90 (11.25)	6.89 (17.78)	14.80 (28.84)
Aalo	4.28 (4.14)	17.65 (20.9)	10.97 (19.98)	32.90 (34.09)

9.3 Scalability

Finally, we evaluate the scalability of PHILAE by comparing its performance with Aalo on a 900-node cluster. To drive the evaluation, we derive a 900-port trace by replicating the FB trace 6 times across ports, *i.e.,* we replicated each job 6 times, keeping the arrival time for each copy the same but assigning sending and receiving ports in increments of 150 (the cluster size for the original trace). We also increased the

Table 7: [Testbed] Percentage of scheduling intervals where synchronization and rate calculation took more than δ for 150-port and $\delta'(= 6 \times \delta)$ for 900-port runs.

	150 ports	900 ports
PHILAE	1%	10%
Aalo	16%	37%

scheduling interval δ by 6 times to $\delta' = 6\times\delta$.

PHILAE achieved 2.72× (9.78×) speedup in average (P90) CCT over Aalo. The higher speedup compared to the 150-node runs (1.50×) comes from higher scalability of PHILAE. In 900-node runs, Aalo was not able to finish receiving updates, calculating new rates and updating local agents of new rates within δ' in 37% of the intervals, whereas PHILAE only missed the deadline in 10% of the intervals. For 150-node runs these values are 16% for Aalo and 1% for PHILAE. The 21% increase in missed scheduling intervals in 900-node runs in Aalo resulted in local agents executing more frequently with outdated rates. As a result, PHILAE achieved even higher speedup in 900-node runs.

As discussed in§5, Aalo's poorer coordinator scalability comes from more frequent updates from local agents and more frequent rate allocation, which result in longer coordinator CPU time in each scheduling interval. Table 6 shows the average coordinator CPU usage per interval and its breakdown. We see that (1) on average PHILAE spends much less time than Aalo in receiving updates from local agents, because PHILAE does not need updates from local agents at every interval – on average in every scheduling interval PHILAE receives updates from 49 local agents whereas Aalo receives from 429 local agents, and (2) on average PHILAE spends much less time calculating new rates and send new rates. This is because rate calculation in PHILAE is triggered by events and PHILAE did not have to flush rates in 66% of the intervals.

10 Related Work

Coflow scheduling: In this paper, we have shown PHILAE outperforms prior-art non-clairvoyant coflow scheduler Aalo from more efficient learning of coflow sizes online. Saath [30] and Graviton [29] also learn coflow sizes online using priority queues and hence suffers the same inefficiency as Aalo. Graviton [29] uses the number of ports a coflow is present at, as an additional indicator of its size. In [19], Aalo was shown to outperform previous non-clairvoyant coflow schedulers Baraat [24] by using global coordination, and Orchestra [20] by avoiding head-of-line blocking.

Clairvoyant coflow schedulers such as Varys [21] and Sincronia [7] assume prior knowledge of coflows upon arrival. Varys runs a shortest-effective-bottleneck-first heuristic for inter-coflow scheduling and performs per-flow rate allocation at the coordinator. Sincronia improves the scalability of the centralized coordinator of Varys by only calculating the coflow ordering at the coordinator (by solving an LP) and offloading flow rate allocation to individual local agents. Sincronia is orthogonal to PHILAE; once coflow sizes are learned through sampling, ideas from Sincronia can be adopted in PHILAE to order coflows and offload rate allocation to local ports. CODA [44] tackles an orthogonal problem of identifying flows of individual coflows online.

However, recent studies [19, 40] have shown various reasons why it is not very plausible to learn flow sizes from applications beforehand. For example, many applications stream data as soon as data are generated and thus the application does not know the flow sizes until flow completion, and learning flow sizes from applications requires changing either the network stack or the applications.

Flow scheduling: There exist a rich body of prior work on flow scheduling. Efforts to minimize flow completion time (FCT), both with prior information (*e.g.,* PDQ [26], pFabric [9]) and without prior information (*e.g.,* Fastpass [35], PIAS [13], [14]), fall short in minimizing CCTs which depend on the completion of the last flow [21]. Similarly, Hedera [8] and MicroTE [15] schedule the flows with the goal of reducing the overall FCT, which again is different from reducing the overall CCT of coflows.

Speculative scheduling Recent works [16, 33] use the idea of online requirement estimation for scheduling in datacenter. In [31], recurring big data analytics jobs are scheduled using their history.

Job scheduling: There have been much work on scheduling in analytic systems and storage at scale by improving speculative tasks [11, 12, 43], improving locality [10, 41], and end-point flexibility [17, 38]. The coflow abstraction is complimentary to these work, and can benefit from them. Combining coflow with these approaches remains a future work.

Scheduling in parallel processors: Coflow scheduling by exploiting the spatial dimension bears similarity to scheduling processes on parallel processors and multi-cores, where many variations of FIFO [37], FIFO with backfilling [32] and gang scheduling [25] have been proposed.

11 Conclusion

State-of-the-art online coflow schedulers approximate the classic SJF by implicitly learning coflow sizes and pay a high penalty for large coflows. We propose the novel idea of sampling in the spatial dimension of coflows to explicitly and efficiently learn coflow sizes online to enable efficient online SJF scheduling. Our extensive simulation and testbed experiments show the new design offers significant performance improvement over prior art. Further, the sampling-in-spatial-dimension technique can be generalized to other distributed scheduling problems such as cluster job scheduling. We have made our simulator publicly available at https://github.com/coflowPhilae/simulator [6].

Acknowledgement We thank our shepherd Patrick Stuedi and the anonymous reviewers for their insightful comments.

References

[1] Apache hadoop. http://hadoop.apache.org.

[2] Apache spark. http://spark.apache.org.

[3] Apache tez. http://tez.apache.org.

[4] Coflow trace from facebook datacenter. https://github.com/coflow/coflow-benchmark.

[5] Microsoft azure. http://azure.microsoft.com.

[6] Philae simulator. https://github.com/coflowPhilae/simulator.

[7] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 16–29, New York, NY, USA, 2018. ACM.

[8] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[10] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[12] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 265–278, Berkeley, CA, USA, 2010. USENIX Association.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 25:1–25:7, New York, NY, USA, 2014. ACM.

[14] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, Oakland, CA, 2015. USENIX Association.

[15] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[16] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 239–252, New York, NY, USA, 2017. ACM.

[17] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 231–242, New York, NY, USA, 2013. ACM.

[18] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[19] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 393–406, New York, NY, USA, 2015. ACM.

[20] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[21] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[22] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. volume 51, pages 107–113, New York, NY, USA, January 2008. ACM.

[24] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 431–442, New York, NY, USA, 2014. ACM.

[25] Dror G. Feitelson and Morris A. Jette. Improved utilization and responsiveness with gang scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 238–261, London, UK, UK, 1997. Springer-Verlag.

[26] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.

[27] Xin Sunny Huang, Xiaoye Steven Sun, and T.S. Eugene Ng. Sunflow: Efficient optical circuit scheduling for coflows. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 297–311, New York, NY, USA, 2016. ACM.

[28] 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.

[29] Akshay Jajoo, Rohan Gandhi, and Y. Charlie Hu. Graviton: Twisting space and time to speed-up coflows. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[30] Akshay Jajoo, Rohan Gandhi, Y. Charlie Hu, and Cheng-Kok Koh. Saath: Speeding up coflows by exploiting the spatial dimension. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 439–450, New York, NY, USA, 2017. ACM.

[31] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[32] David A. Lifka. The anl/ibm sp scheduling system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '95, pages 295–303, London, UK, UK, 1995. Springer-Verlag.

[33] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[34] Stanley Lemeshow Paul S. Levy. *Sampling of Populations: Methods and Applications*. Wiley, 4 edition, Jun 2012.

[35] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.

[36] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[37] Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 629–638, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[38] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.

[39] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Process Scheduling. Operating System Concepts.* John Wiley & Sons, 8 edition, 2010.

[40] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, Boston, MA, 2019. USENIX Association.

[41] 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 *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[43] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[44] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 160–173, New York, NY, USA, 2016.

.

PostMan: Rapidly Mitigating Bursty Traffic by Offloading Packet Processing

Panpan Jin[1], Jian Guo[1], Yikai Xiao[1], Rong Shi[2], Yipei Niu[1], Fangming Liu[1*], Chen Qian[3], Yang Wang[2]
[1]*National Engineering Research Center for Big Data Technology and System,*
Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]*The Ohio State University, USA*
[3]*University of California Santa Cruz, USA*

Abstract

Unexpected bursty traffic due to certain sudden events, such as news in the spotlight on a social network or discounted items on sale, can cause severe load imbalance in backend services. Migrating hot data—the standard approach to achieve load balance—meets a challenge when handling such unexpected load imbalance, because migrating data will slow down the server that is already under heavy pressure.

This paper proposes PostMan, an alternative approach to rapidly mitigate load imbalance for services processing small requests. Motivated by the observation that processing large packets incurs far less CPU overhead than processing small ones, PostMan deploys a number of middleboxes called helpers to assemble small packets into large ones for the heavily-loaded server. This approach essentially offloads the overhead of packet processing from the heavily-loaded server to others. To minimize the overhead, PostMan activates helpers on demand, only when bursty traffic is detected. To tolerate helper failures, PostMan can migrate connections across helpers and can ensure packet ordering despite such migration. Our evaluation shows that, with the help of PostMan, a Memcached server can mitigate bursty traffic within hundreds of milliseconds, while migrating data takes tens of seconds and increases the latency during migration.

1 Introduction

Modern distributed systems usually scale by partitioning data and assigning these partitions to different servers [4, 10, 11, 14, 15, 17, 33, 37, 41, 50]. In such an architecture, some servers may experience a higher load than others, creating a classic load imbalance problem [11, 16].

Many works have studied how to mitigate load imbalance by better data partitioning and placement strategies [4, 10, 11, 14, 16, 17, 41], which work well for long-term and stable load imbalance. For load imbalance caused by unexpected bursty traffic, however, these approaches meet an additional challenge: to adapt to such unexpected load imbalance, we need to adjust the data partitioning or placement strategies online by migrating hot data to less busy servers, but migrating hot data will inevitably slow down the server hosting hot data—this is the server we want to accelerate. This means to alleviate load imbalance, these approaches will first exacerbate load imbalance for a while, which is a risk that production systems are often unwilling to afford. For example, Amazon Dynamo runs data migration at the lowest priority, and finds that during a busy shopping season, data migration can take almost a day to complete [16]. Unfortunately, unexpected bursty traffic is frequently reported in practice, for various reasons such as a sudden event drawing public attention on a social network [9] or a hot item on sale [18, 34, 39, 40].

This paper proposes PostMan, an alternative approach to mitigate load imbalance for services that are processing small packets, which usually incur a high overhead for packet processing. Typical examples of such services include key-value stores and metadata servers. For example, Facebook reported that in its caching layer, most key sizes are under 40 bytes and median value size is 135 bytes [5, 38]; metadata servers, such as NameNode in HDFS [50], are usually processing packets with a size of tens to a few hundred bytes.

The key idea of PostMan is motivated by the observation that there is a significant gap between the overhead of processing small and large packets, because the networking stack has to pay a constant overhead for each packet, such as interrupt handling and system call. For example, when processing 64B packets on 10Gb Ethernet, Linux can achieve a throughput of 2.4 Gbps with a CPU utilization of 800%. On the contrary, when processing 64KB packets, Linux can achieve a throughput of 9.1Gbps with a CPU utilization of only 220%. Newer networking stacks, such as mTCP [24] and IX [7], have mitigated this problem, but first, the gap still exists, though smaller, and second, a wide deployment of a new networking stack requires a big effort, because the networking stack is a critical component of the whole system, which may affect all

*The corresponding author is Fangming Liu (fmliu@hust.edu.cn).

other components.

Motivated by this observation, PostMan incorporates a number of middleboxes called helpers, which batch small packets for the server experiencing bursty traffic (called "helpee" in the rest of the paper), so that the helpee can enjoy the low overhead of processing large packets. This approach essentially offloads the constant overhead associated with each small packet from the helpee to the helpers.

This approach brings several benefits: first, PostMan does not require the time-consuming data migration. Instead, it only requires the clients to re-connect to the helpers, which can be completed within hundreds of milliseconds as shown in our evaluation. Second, PostMan can incrementally deploy new networking stacks on helpers and allow other servers to still use traditional stacks. Third, PostMan can use multiple helpers to accelerate one helpee, which means its capability is not limited by the power of a single machine. Finally, offloading batching opens up new opportunities for further optimization: we observe that packets to the same destination have significant redundancy in their packet headers (e.g., same destination IP and port). By removing such redundancy, PostMan is able to achieve a considerable reduction in bandwidth consumption at the helpee.

Of course, PostMan has its own limitation: if load imbalance lasts long, PostMan will be more expensive than data migration because incorporating helpers incurs additional data transfer and extra server resource. Therefore, we expect PostMan and data migration to be complementary to mitigate load imbalance: for unexpected bursty traffic, we can activate PostMan to accelerate the heavily-loaded server first; if such burst continues to happen regularly, we can migrate data when the machine is less busy; after data migration is completed, we can disable PostMan to minimize cost. As a result, the helpers would not be active for a long time. Moreover, since PostMan only targets the servers experiencing bursty traffic, we can use a few helpers for a large cluster to further reduce cost.

The idea of batching small requests to improve performance is certainly not novel. The key novelty of PostMan lies in its observation that, *for the purpose of mitigating unexpected load imbalance, batching should be performed remotely and on demand*: remote batching allows PostMan to accelerate a heavily-loaded server with the help of resource from other servers; on-demand batching allows PostMan to minimize the overhead by only helping those servers experiencing bursty traffic. To realize these ideas, PostMan includes three main components:

- We provide an efficient implementation of the helpers. By utilizing state-of-the-art techniques like DPDK and efficiently parallelizing work among multiple cores, a single helper node can assemble and disassemble around 9.6 million small packets per second. By removing redundancy in headers, PostMan can reduce packet header size from 46 bytes to 7 bytes: for 64-byte packets, this means about 50% higher bandwidth utilization at the helpee.
- To ensure packet ordering despite migrating connections across helpee and helpers and despite helper failures, PostMan keeps helpers *stateless* by maintaining sufficient information at the clients and servers to detect out-of-order packets and retransmit packets when necessary. While we find many applications already implement similar functionalities, we provide a library to those which do not.
- We present an adaptive batching mechanism to decide how many packets to assemble. It can increase batch size for higher throughput under heavy traffic and decrease batch size for lower latency under light traffic.

Our evaluation on Memcached and Paxos shows that, with the help of PostMan, the service can mitigate bursty traffic within hundreds of milliseconds, while migrating data can take tens of seconds. Further investigation shows that this is because PostMan can improve the goodputs of Memcached and Paxos by 3.3× and 2.8×, respectively.

2 Related work

Load balancing. Load balancing is a classic topic of distributed systems. Most existing systems use an adaptive approach to achieve load balancing: they can monitor the load of each machine and place new data on less busy machines [4, 10, 11, 14, 16, 17, 20, 41]; some of them can split, merge, and migrate existing data partitions online (e.g., Bigtable [11], RAMCloud [41] and ElMem [20]).

Despite the support from such mechanisms, how to mitigate load balancing caused by unexpected bursty traffic is still a challenge: since these mechanisms will put more pressure on the machine that is already heavily loaded, the administrator is facing a painful trade-off between the short-term loss and the long-term gain of migrating hot data.

Batching small packets. A classic method to improve the performance of processing small packets is to batch them to amortize the constant overhead across multiple packets. For example, TCP has the Nagle's algorithm to batch small packets. Modern NICs often use Generic Receive Offload (GRO) to re-segment the received packets. However, the power of such per-connection batching mechanisms is limited by the number of outstanding packets per client. In many cases, a client may have to wait for replies before it can issue new ones and thus the number of packets that can be batched is limited.

Comet [21] batches the received data before batched stream processing at the server side. KV-direct [29] first batches multiple KV operations at the client side to increase bandwidth utilization, and then at the server side, it batches memory accesses by clustering computation together.

A few systems incorporate a number of nodes to batch packets for other nodes. For example, Facebook has built mcrouter to batch packets for its Memcached service [38]. NetAgg aggregates traffic along network traffics for applications following a partition/aggregation pattern [35]. MPI has a collective I/O mode to batch I/Os from multiple processes before writing them to disks [36].

Compared to these systems, PostMan uses batching for a different goal—mitigating unexpected load imbalance. To achieve this goal, PostMan offloads the overhead of batching from the heavily-loaded server to others and only performs such offloading when a server is under heavy pressure. These techniques allow PostMan to use extra resource to accelerate a server experiencing bursty traffic and minimize the overhead when there is no such bursty traffic.

Efficient network stack. There is a continuous effort to develop more efficient network stacks for high-speed networks: mTCP [24] moves the TCP stack to the user space to reduce system call overhead and further improves performance by batching I/Os; DPDK [1] asks a network card to transfer data to the user space directly and applies a series of optimizations like CPU affinity, huge page, and polling to get close to bare-metal speed; IX [8] and Arrakis [43] design new operating systems to separate data transfer and access control to achieve both high speed and good protection; Netmap [46] improves networking performance by reducing memory allocation, system call, and data copy overhead; many works [12, 22, 25, 26, 30–32, 44, 48, 49, 51] exploit the RDMA technique and FPGA to improve networking performance.

Although these works have significantly improved the network performance, the performance gap between small and large packets persists (Table 1). Taking IX [8] as an example, it can achieve almost 10Gbps bandwidth even with 64-byte packets, which is significantly better than Linux. However, first, it needs to consume a considerable amount of CPU resource (see Section 6.2); second, there is still a gap between goodput (bandwidth used for payload) and throughput, because packet headers consume a large portion of bandwidth. Such per-packet overhead exists in RDMA systems as well [27].

Furthermore, the deployment of new networking stacks is usually a slow procedure, because networking service is critical and production systems are unwilling to pay any risk. On the other hand, PostMan allows administrators to incrementally deploy such new techniques on a few servers to accelerate a large number of legacy servers.

Others. The architecture of PostMan is similar to existing proxies (e.g., NGINX [45] and mcrouter [38]), which are also deployed between clients and servers. The key difference is that PostMan dynamically enables and disables helpers according to the load of servers.

The design of PostMan may seem to be similar to that of the split TCP approach [19, 42, 47], which also deploys helper nodes in the network. However, their goals and internal mechanisms are totally different: split TCP is designed to reduce latency in a network with large round-trip delays, by letting helper nodes send *ack*s to the sender directly; PostMan, on the other hand, is designed to improve the throughput of transferring small packets by letting helper nodes batch small packets. For the purpose of tolerating helper failures, PostMan's helpers actually delay sending *ack*s to the sender, until the helpers receive *ack*s from the helpee (see Section 4.2).

	64 bytes	64KB
10Gb Linux	2.4Gpbs	9.1Gbps
10Gb IX [8]	5.0Gbps	9Gbps

Table 1: Goodput of processing big and small packets. Goodput excludes bandwidth used for headers.

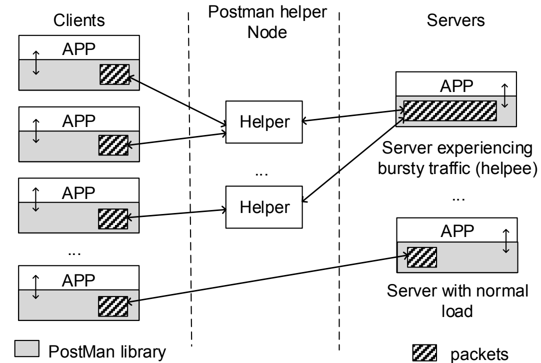


Figure 1: Overview of PostMan.

3 Overview of PostMan

In this paper, we propose PostMan, a distributed service to offload the overhead of packet processing from a heavily-loaded server (called helpee in the rest of this paper). Motivated by the observation that processing large packets incurs far less overhead than processing small packets, PostMan deploys a number of helper nodes in the network to assemble small packets for the helpee. By doing so, PostMan essentially offloads the constant overhead associated with each packet from a helpee to the helpers. With the help of PostMan, a helpee node only needs to process large packets.

Figure 1 shows the organization of PostMan. The core of PostMan consists of: 1) a number of helper nodes that assemble small packets for the helpees, and 2) a PostMan library that provides the applications with the functionalities of packet re-direction, assembly, and disassembly. Furthermore, PostMan library provides functions to detect out-of-order packets and re-transmit packets when necessary. These functions allow PostMan to achieve fault tolerance and load balance by migrating connections across helpers.

As shown in Figure 1, in a large scale distributed system,

PostMan will only activate helpers to accelerate servers experiencing unexpectedly high load, which causes their latency to be higher than their service level agreement (SLA). For servers with normal load, their clients should communicate with the servers directly. Accelerating these normal servers with PostMan, though possible, is not cost effective. Essentially PostMan *offloads* overhead instead of *reducing* overhead: in fact, PostMan increases overall overhead because it needs to perform additional work to assemble and disassemble packets. Therefore, PostMan tries to keep such overhead low by only helping nodes with trouble.

4 PostMan Design

PostMan is designed for the scenario that, suddenly, a large number of clients are sending small requests to a few servers (i.e., helpees). PostMan deploys helper nodes to assemble the clients' small packets to the helpee and disassemble the helpee's small packets to the clients, so that the helpee only needs to process large packets. To differentiate these two directions, we use "request" to refer to a packet from a client to a server and "reply" to refer to a packet from a server to a client.

In the rest of this section, we discuss how to assemble and disassemble packets efficiently at helper nodes, what APIs PostMan provides and how to apply them, and how to adaptively balance throughput and latency.

4.1 Assembling and disassembling packets

Format of assembled packet: For small packets, the size of their headers (at least 20 bytes for IP and TCP header respectively, 6 bytes for MAC header) is comparable to the size of their payloads, and that is one major reason why network throughput cannot reach bare-metal bandwidth, even with new techniques like DPDK. However, when considering packets to the same destination, their headers contain a significant amount of redundancy: packet assembly at the helper nodes can remove such redundancy and further improve throughput at the helpee. For example, since packets to be assembled have the same destination, PostMan only needs to maintain one copy of destination IP and port in the assembled packet. PostMan can shrink source IP as well for small to medium clusters by maintaining a mapping from IP to a shorter identification number at each node (e.g., 2 bytes for clusters with less than 64K machines). Moreover, since the connections from the clients to the helper and the connections from the helper to the helpee are separate, they perform congestion control independently, which means the helper can simply discard related information in the original packets.

As shown in Figure 2, a helper node can assemble packets from multiple connections, and when doing so, the helper discards their TCP/IP/MAC headers and only sends their payloads, together with one TCP/IP/MAC header for all payloads,

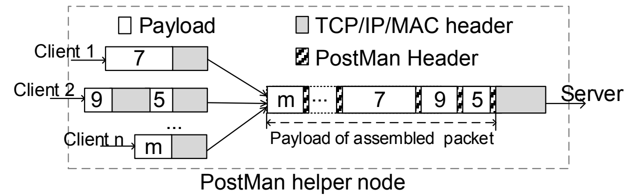


Figure 2: PostMan assembles packets from different clients by using a short header for each payload.

Type	Length (bytes)	Description
ID code	1	Message type
Len	2	Message length
Sender	2* + 2	Src IP/port

Table 2: The format of PostMan header. (*: for a cluster with less than 64K machines, the helper extracts the lower 16 bits from a source IP and then hashes them into a 2-byte identifier)

so that the helpee does not need to pay the header overhead for each packet. However, to ensure that the helpee can correctly disassemble packets, the helper node must encapsulate necessary information for each payload, which is called a "PostMan header".

A PostMan header is a 3-tuple structure (Table 2): 1) an identification code to identify the packet type, 2) a length field to record the length of one payload, and 3) the source IP and port of the payload to locate the sender. A packet can have one of the following three types: 1) *request*, i.e., a packet sent by a client, 2) *reply*, i.e., a packet sent by a server, and 3) *connect*, i.e., a command to create a connection (see Section 4.2). As a result, compared to a TCP/IP/MAC header that takes at least 46 bytes, a PostMan header only takes 7 bytes: this is a significant saving when processing small packets.

Workflow of assembling and disassembling packets: When assembling packets from the clients to the helpee, a helper node fetches all pending packets in its network stack, replaces their TCP/IP/MAC headers with corresponding PostMan headers, concatenates all of them, and adds its own TCP/IP/MAC header. By doing so, both the PostMan headers and the payloads of the original small packets become the payload of the assembled packet.

On the opposite direction, when a helpee sends replies to clients, it will first send the assembled reply to the helper node, which will disassemble the replies and dispatch them to the clients. The format of the assembled reply is similar to that in Figure 2.

Each helper node can create multiple connections to the helpee, so that one helpee can utilize multiple cores and threads to receive packets concurrently.

4.2 PostMan library

A traditional networking library provides a number of APIs, such as *bind*, *connect*, *send*, and *recv* to the application. PostMan library provides a few additional ones: *pm_connect* allows a client to create a connection to a helper node; *compose* and *decompose* assemble and disassemble packets as described in Section 4.1; *get_info* allows the application to retrieve connection information. A developer should use these additional APIs together with traditional APIs to build the application. Next, we show how these functions work and how to modify an application to utilize these functions.

Establish connections. A server should bind to a port and wait for new connections, like using a traditional network library. Of course here a server may accept new connections from the helper nodes. A client can use the traditional network library when the latency is low and switch to PostMan when the latency is high by calling *pm_connect*. *pm_connect* will choose a helper (see Section 5.1) and connect to the helper. Then it sends a special "connect" packet to the helper node. This packet contains the destination IP and port of the helpee and the source IP and port of the client. The helper node will connect to the corresponding helpee, if there is no connection yet, and forward this packet. At the same time the helper creates a mapping from the client's socket to the server's socket. When the server application reads data from a helper connection, it should call *decompose*, which will identify the special "connect" packet and notify the server application that a new client tries to connect. Finally, the server library will return a "success" packet to the client through the helper node, telling the application *pm_connect* succeeds.

Transfer data. When PostMan is activated, a client should send packets through the connection to the helper. The helper node assembles multiple small packets and sends the assembled packet to the server. When the server application reads a packet, it calls *decompose* to disassemble the packet into small packets and process them. On the opposite direction, when a server sends replies, it should buffer multiple replies and assemble them by calling *compose*. Then it can send the assembled reply to the connection to the helper. The helper node disassembles the replies and sends them to the corresponding clients based on the PostMan headers. The clients can read packets using a *recv* or *read* system call.

Ensure packet ordering. Applications often need to ensure packets are not lost, duplicated, or re-ordered. Since PostMan uses TCP connections between the clients and the helper nodes, and between the helper nodes and the helpees, one can easily verify that these properties hold when there is no migration of connections. However, PostMan may migrate connections for several reasons: if a client is connecting directly to a server and finds the latency is high, it will call *pm_connect* to migrate its connection to a helper; when a helper is heavily loaded, PostMan will instruct its clients to migrate to other helpers; finally if a helper fails, its clients must migrate to other helpers as well. As a result, PostMan needs additional mechanisms to ensure packet ordering despite such connection migration. For the first two cases, where migration is executed gracefully, a simple solution is to ask a client to wait for replies of all its pending requests before migrating its connection. For the helper failure case, however, this problem becomes challenging, because a client is uncertain about which packets are delivered.

In distributed systems, two approaches are widely used to achieve fault tolerance: one is to replicate the nodes that can be faulty, and the other is to re-direct requests to another node. Replication can fully hide faults from upper layers, at the cost of increased overhead. The re-direction approach has lower overhead, but it requires the faulty node to be *stateless*, i.e., it does not have any important state that will affect execution.

We use the re-direction approach because of its low overhead. To ensure packet ordering even if the system loses all information on the faulty helper node, PostMan needs to maintain sufficient information at the senders and receivers. Its basic idea is similar to that of TCP: a sender should buffer a packet until it gets acknowledged and re-send a packet if it does not get acknowledgement in a given amount of time; a receiver should check the sequence numbers in the packets to ensure they are in order. Unlike TCP that implements this mechanism in one connection, PostMan needs to implement this mechanism across different connections because when a helper fails, the client needs to re-connect to a new helper.

On one hand, we observe that many applications have already implemented such mechanism. The fundamental reason they choose to do so instead of relying on TCP is that they are designed to tolerate machine failures: in this case, a node needs to connect to other nodes and face the same problem as PostMan. For these systems, PostMan can utilize the application's mechanism directly.

On the other hand, for applications that do not have this mechanism, PostMan provides a general solution. To ensure packets will not be lost, PostMan library at senders[1] will buffer packets until it receives *acks* from receivers, like the TCP protocol does. Since the underlying layer maintains separate TCP connections between the senders and the helper nodes, and between the helper nodes and the receivers, the key to avoid data loss is to coordinate the underlying *ack* mechanisms: after receiving a packet from a sender, the helper node should not send the *ack* to the sender until it has got *ack* from the receiver. We modify the TCP implementation at the helper nodes to realize this mechanism. Since Postman targets small packets, delaying acks and sending them in burst should have little impact on congestion control.

When a helper fails, a sender may not receive *acks* for its outgoing packets, so it may decide to reconnect to another helper and retransmit those packets through the new helper.

[1] Note that senders and receivers are different concepts compared to clients and servers: when a server is receiving packets from a client, it is the receiver; when a server is sending a reply to a client, it is the sender.

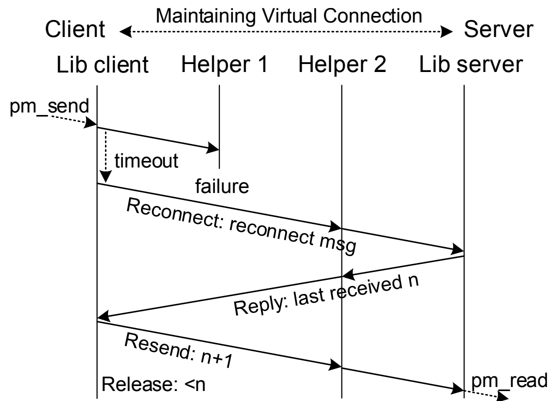


Figure 3: Maintaining a virtual connection by redirecting and re-sending requests when a helper node fails.

In this case, since the receiver may have already received these packets (*acks* may be lost due to helper failure), these packets may be duplicated or re-ordered. To prevent such abnormality, PostMan libraries at the sender and the receiver maintain additional information to detect duplicate or out-of-order packets.

To be specific, the library will keep track of how many bytes are already sent and received on each connection; for each buffered outgoing packet, the library will record its offset in the stream. As shown in Figure 3, when a helper node fails, the client library, which is the sender in this example, will connect to another helper node and sends a "reconnect" message to the server through the new helper node, which contains the number of sent and received bytes at the client side. When the server library receives this command, it will first stop receiving packets from the old connection and then respond with the number of sent and received bytes at the server side. By exchanging the number of sent and received bytes and comparing them to the offsets of buffered packets, both sides can determine which packets should be re-transferred.

Further optimization. So far we assume an application server needs to disassemble packets before processing them. However, this may not be necessary for some applications. A typical example is a server that needs to forward or broadcast packets (e.g., proxy server, leader replica in replication protocols, etc). For such servers, since they do not care about the content of payload, they can forward or broadcast the assembled packets directly, instead of disassembling them first. Note that when sending assembled packets, the application should not use PostMan, since these packets are large.

Using PostMan library. To apply PostMan to existing applications, the developer needs to modify its code: at the client side, the client should call *pm_connect* to switch to PostMan when it observes a high latency and switch back to traditional sockets when the latency drops back to normal; at the server side, the server should call *decompose* when it receives a packet from a helper (*get_info* can tell whether a connection is from a client or from a helper) and should assemble a number of replies by calling *compose* when it sends packets through helpers. If the application needs PostMan's help to ensure packet ordering, it should notify PostMan when a packet is sent or received, so that PostMan can buffer and release packets and update corresponding metadata.

It is possible to hide all the mechanisms mentioned above in the library and provide the applications with an illusion that they are using direct connections between clients and servers. We have implemented a library to achieve such transparency. However, we find it can incur up to 50% overhead for additional operations like memory copy, synchronization, context switch, etc. Considering the main goal of this work is to improve the performance of the heavily-loaded server, we decide to give up transparency for better performance.

4.3 Adaptive batching

Batching can affect system latency in two opposite ways: on one hand, to assemble packets, a helper node must wait for a certain amount of time to accumulate enough small packets, which will increase the latency of the system. On the other hand, according to queuing theory, when the load is close to or higher than the system's capacity, queuing delay will become a dominant factor for latency. Since batching can improve a system's capacity, it can reduce queuing delay and thus can reduce latency.

PostMan partially avoids such trade-off by only activating helpers for heavily-loaded servers. In addition, PostMan incorporates an adaptive batching algorithm to balance latency and throughput when helpers are enabled. Like many systems using batching, PostMan defines a maximum batch interval (T) and a preferred batch size (S): if the helper has waited for T (condition 1) or if its assembled packet has reached size S (condition 2), the helper will send the assembled packet to the helpee. Then the question turns to how to set T and S: large T and S lead to unnecessary waiting when traffic load is light; small T and S reduce the chance of assembling packets when traffic load is heavy.

To address this problem, PostMan uses an adaptive batch size and interval to increase throughput under heavy loads and decrease latency under light loads, as shown in Algorithm 1. PostMan records the batch size (s) and waiting time (t) of the last batch: if s is significantly different from S or t is significantly different from t, PostMan updates S and T accordingly. Furthermore, it sets a lower bound of S and T to ensure efficiency. Note that although T is the maximum batch interval, the actual interval t can be much larger than T when the helper does not receive any packets for a long time; s can be much larger than S as well when the helper receives many packets at the same time.

This algorithm has a few parameters: we set the lower bound of S to be 1500 because that is the MTU size; the lower

Algorithm 1 Adaptive batching algorithm

Input: the size (s) and waiting time (t) of last batch

```
 1: procedure Update S and T
 2:     S_l ← 0.75S
 3:     S_u ← 1.25S
 4:     if (s < S_l ∨ s > S_u) ∧ (s ≥ 1500) then
 5:         S ← s
 6:     end if
 7:     T_l ← 0.5T
 8:     T_u ← 1.5T
 9:     if (t < T_l ∨ t > T_u) ∧ (t ≥ 10µs) then
10:         T ← t
11:     end if
12: end procedure
```

bound of T should be set according to the SLA requirement; we set other parameters based on empirical experiments.

5 Implementation

In this section, we present how to achieve efficiency and scalability for PostMan.

5.1 Efficient helper

Fast I/O and user-level stack: Each helper node needs to assemble requests from the clients, and disassemble the replies from the servers. To efficiently process the small packets on the helper nodes, we implement PostMan upon DPDK [1], which is a set of libraries and drivers for fast packet processing. DPDK minimizes the overhead of packet processing by transferring packets from NICs directly to user space programs and thus can achieve a throughput of hundreds of millions packets per second. Upon DPDK, we use mTCP [23] to handle TCP protocol and connections. DPDK provides a poll mode I/O model, which can transfer a batch of packets in one I/O operation. This I/O model not only avoids the overhead caused by frequent interrupts in per-packet based processing in Linux, but also naturally fits the assembling requirements of our helper nodes: a helper can simply add all the payload data from these packets to the assembling buffer, instead of performing the read operation several times.

In-stack processing: Since the assembling logic only involves simple operations for request/response headers, PostMan implements these operations in the network layer to accelerate the identification and pre-processing of the original packets. PostMan uses direct data exchange between the server and the client streams so as to avoid redundant memory copy and improve the performance of fragmented data operation. By implementing everything in the network stack, PostMan eliminates the interaction and context switching between the applications and the stack to further improve assembling efficiency. All necessary assembling and disassembling operations are queued in the stack, so that PostMan can perform them after finishing processing the incoming packet in the TCP protocol. Furthermore, PostMan only keeps the necessary procedures for receiving and sending packets in a TCP stream. Other operations, like the ICMP protocol, are abandoned, either because they have nothing to do with packet assembly, or they should be performed in the helpee's stack.

Independent per-core context: PostMan leverages per-core thread (affiliated to a hardware thread) and independent per-core context to avoid synchronization among different assembling threads. On each helper node, we enable the Receive Side Scaling (RSS) [3] function of NIC—this is widely supported by today's NICs—to hash flows into different physical queues in hardware, where each queue is assigned to a dedicated CPU thread. PostMan does not share any global information, hence the connection can be locally processed on each core. PostMan sets up at least one queue for each thread, such that the flows in each RSS group can be processed independently without exchanging any information between CPU cores. Consequently, PostMan can scale well on today's multi-core system. Note that RSS will reduce a helper's chance to batch packets, but since PostMan is enabled only when the server is under high load, there are still plenty of chances for a helper to batch packets as shown in our evaluation. For each thread, PostMan uses hugepages to store raw packet data, similar as many DPDK based applications, so as to reduce the number of TLB misses; PostMan uses hardware-based CRC instruction for flow hashing to accelerate the assembling process.

5.2 Scalability and load balancing

As presented in the previous section, PostMan is designed for hardware thread and RSS built-in NIC. Hence, PostMan scales well with the number of cores.

For helper nodes, their stateless nature, which allows connections to be migrated freely across helpers, significantly simplifies scaling and load balancing: when a client connects to helpers, it picks up one with a low load; whenever some helper nodes are overloaded, they can simply disconnect some clients and those clients will automatically re-connect to other helper nodes. To achieve this, PostMan uses a standard load-balancing technique: each helper monitors its own CPU and network utilization; when a client establishes a connection, either for a new connection or for re-connecting, it randomly chooses a number of helpers; each helper will reply with its resource utilization, so that the client can choose the helper with the lowest utilization; when a helper finds its resource utilization is too high, it disconnects existing connections, so that the corresponding clients can connect to other helpers.

PostMan has no inherent scalability bottleneck: since a client can connect to any helper, PostMan does not need a centralized master node to map clients to helpers.

5.3 When to enable and disable helpers?

PostMan provides a mechanism to enable and disable helpers on demand, but in practice, we still need to answer the policy question about when to enable and disable helpers. In principle, both the clients and the servers can make the decision and we observe the following trade-offs:

A client can monitor its perceived latency to the server and make decisions accordingly: this approach brings minimal overhead to the server side, but since a client cannot gain the overall load statistics of the server, it may not be able to make the best decisions in certain cases. On the other hand, a server certainly has more information to make a better decision, but to execute the decision, the server must pay the overhead of notifying corresponding clients and helpers, which could be problematic if the server is already under heavy load.

Therefore, the current implementation of PostMan uses a hybrid approach: a client will decide to enable helpers (i.e., the client disconnects from the server and connects to a helper) if it detects its perceived latency is higher than SLA—this could minimize the overhead at the server side when the server is busy; a server will decide to disable helpers when its throughput becomes low—the overhead is fine in this case since the server is not too busy.

6 Evaluation

The goal of PostMan is to quickly mitigate the bursty traffic directed to one or a few servers. To assess whether PostMan achieves this goal, we evaluate the performance of PostMan using various applications and workloads. In particular, our evaluation answers the following questions:

- How well can PostMan help a service reduce the load caused by bursty traffic?
- How is PostMan's capability affected by packet size?
- How much resource does PostMan need to achieve such benefit?
- How does the system perform when there are helper failures?

To answer the first question, we run benchmarks on Memcached and Paxos, and emulate the case of bursty traffic by drastically increasing the load during the middle of the experiment; we enable PostMan after such burst to measure 1) whether it can reduce the latency of the target service and 2) how long it takes to enable PostMan. We compare the results of PostMan to those of the data migration approach.

To answer the following three questions, we use a ping-pong microbenchmark to measure the performance of PostMan under different parameters.

Memcached. Memcached is a key-value based memory object caching system [2]. It is used widely in the data centers to cache data to speed up the lookups of frequently accessed data. As reported in [5], Memcached is often used to store small but hot data. We have modified 454 lines of code in Memcached 1.4.32 to apply PostMan. The benchmark generates GET/SET commands with a fixed key size (32 bytes) and different value sizes.

Paxos. Paxos is an asynchronous replication protocol to achieve consensus in a network of unreliable processors [28]. Paxos needs $2f+1$ replicas to tolerate f machine crashes and asynchronous events (e.g inaccurate timeout caused by network partitions). A number of systems, such as Megastore [6], Windows Azure [10] and Spanner [14], are using Paxos for fault tolerance and since they use many Paxos groups, one for each data partition, it is possible that a few of them experience bursty traffic.

In Paxos, one replica is elected as leader, and it needs to broadcast the received requests to other non-leader replicas. It is a typical example of applications that do not care about the contents of packets. Therefore, the leader can read the assembled packets from PostMan and broadcast the assembled packets directly. After the non-leader replicas receive the assembled packets, they will disassemble them. Such mechanism can avoid the redundant disassembling and assembling operations at the leader replica, which is the bottleneck in the system. To exploit such opportunities, we implement our own version of Paxos using PostMan and compare it to a vanilla version that reads individual packets from the clients, assembles them, and then broadcasts the assembled packets. For simplicity, we only implement the Paxos protocol in the failure-free case, which is enough to evaluate the performance benefit of PostMan.

Ping-pong benchmark. This benchmark [8] can test network performance with configurable packet sizes. To avoid the inaccuracy caused by TCP merging packets, this benchmark asks each client to perform a ping-pong like communication with the server: the client sends a packet to the server and then waits for the server to send the packet back. By doing so, since a client has only one outstanding packet, TCP has no chance to merge packets.

Experiment Setup. We run all experiments on CloudLab [13] with 15 machines. Each machine is equipped with an Intel Xeon E5-2660 v3 @ 2.60GHz CPU, with 10 physical cores and hyper-threading, and an Intel 82599ES 10-Gigabit NIC. These severs run Ubuntu 16.0.2 LTS with Linux 4.8.0 kernel, and use DPDK 16.07.2 for the helper nodes. For DPDK Poll Mode Driver, we set the batch size to 64, which is the maximum number of packets received/transmitted in each iteration. For Paxos and Ping-pong experiments, we use IX [8], a state-of-the-art network stack built upon DPDK, at the client side, so that we can stress-test the server with a limited number of client nodes. We also add 17 LoC to count

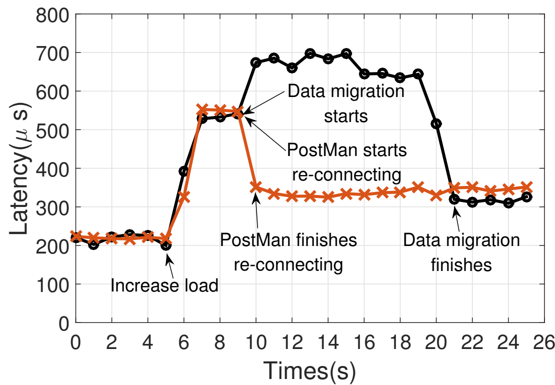


Figure 4: Mitigating bursty traffic in Memcached (PostMan enables two helpers).

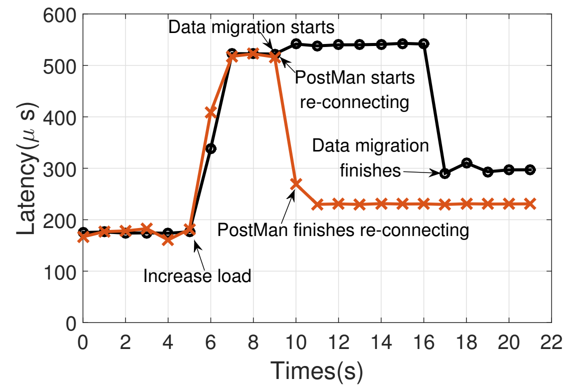


Figure 5: Mitigating bursty traffic in Paxos (PostMan enables two helpers).

the RX/TX bytes and packets in the data plane of IX, whose impact on the performance is negligible in our experiments. For Memcached experiments, since the Linux stack is sufficient to saturate the server, we do not switch to IX. By default, the application server of our benchmarks runs on 16 cores (8 real cores with hyperthreading). Note that in all experiments, when enabling PostMan, our reported goodput does not include the PostMan header and the TCP/IP/MAC header added by the helper nodes, which allows a fair comparison with the goodput without PostMan. When using PostMan, a client enables helpers if its observed 99 percentile latency (p99) is higher than 500 µs [7].

6.1 Effectiveness of PostMan

To measure the effectiveness of PostMan, we emulate the case of bursty traffic on both Memcached and Paxos.

Memcached. We measure p99 latency of Memcached using a read workload with 32B keys and 64B values. As shown in Figure 4, we first use a light load, which incurs a latency of 200 µs, till time 5. Then we increase the load drastically, which increases the latency to more than 500 µs. At about time 9, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 660 client connections and finishes within 550 ms. Afterwards, the latency is reduced to around 300 µs.

As a comparison, we emulate the data migration approach by assuming 50% of the clients are accessing 10% of the data (i.e., 6.4GB) in the Memcached server. Therefore, we start a thread in the Memcached server to copy the data to another server at time 9. Normally such a thread needs to access the internal data structure of Memcached, which may incur additional CPU overhead and may contend with the client's requests. Our emulation avoids such overhead by letting the migration thread copy dummy data to another server, which is in favor of the data migration approach. After data copy is complete, we remove half of the clients since they should be re-directed to another server. As one can see, the data migration takes about 13 seconds and during this procedure, the latency of the service further increases, because the migration traffic and the client's traffic compete for resource. One can of course limit the rate of migrating data to reduce such interference, but that will further increase the length of data migration.

Paxos. We run a similar set of experiments on Paxos. We measure the p99 latency of Paxos using a workload with 64B requests (Paxos does not execute the request, so the content of the request does not matter). As shown in Figure 5, we first use a light load, which incurs a latency of 200 µs, till time 5. Then we increase the load, which increases the latency to about 500 µs. At about time 9, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 960 client connections and finishes within 750 ms. Afterwards, the latency is reduced to around 220 µs.

Similarly, we emulate the data migration approach by assuming 50% of the clients are accessing 10% of the data. Therefore, we start a thread in the non-leader server to copy the data to another server at time 9. Since the leader has the highest overhead in Paxos, copying data from a non-leader server should incur less interference on the clients' requests. After data copy is complete, we remove half of the clients since they should be re-directed to another server. As one can see, the data migration takes about 8 seconds. During this procedure, unlike the Memcached experiments, data migration has little impact on the clients' requests, because it is performed from a non-leader server. Note that after mitigating bursty traffic, the performance of the two systems is different because they have different workloads: with PostMan, the server is processing full load with big packets; after data migration, the server is processing half load with small packets. Which one has better performance depends on the actual workload: in Figure 5 the system with PostMan has lower latency while in Figure 4 the system with PostMan has slightly higher latency.

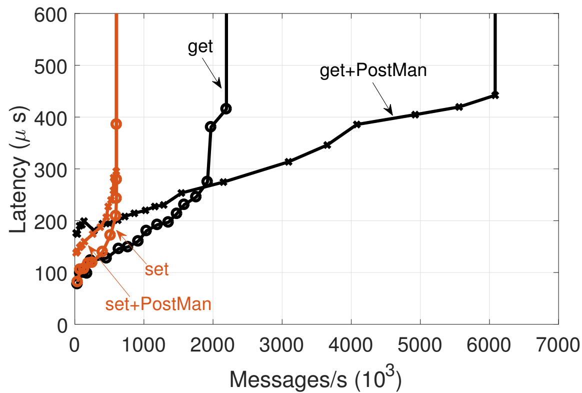


Figure 6: The latency with different load for Memcached and Memcached + PostMan (64-byte payloads; PostMan enables up to five helper nodes).

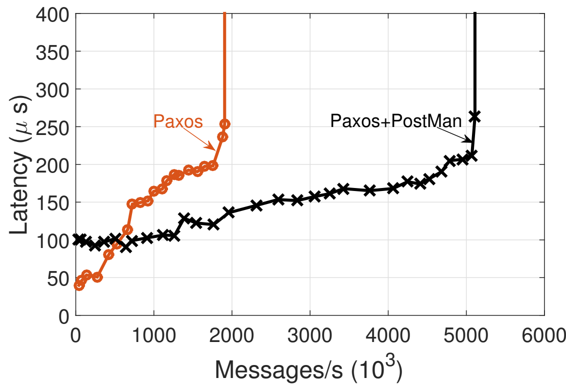


Figure 7: The latency with different load for Paxos and Paxos + PostMan (64-byte payloads; PostMan enables up to six helper nodes).

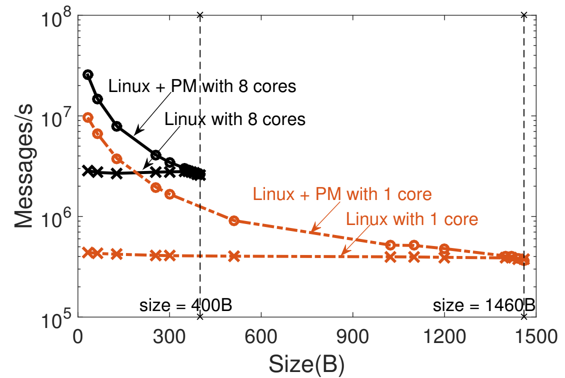


Figure 8: The throughput with different payload sizes for Linux and Linux + PostMan (PostMan enables up to six helper nodes).

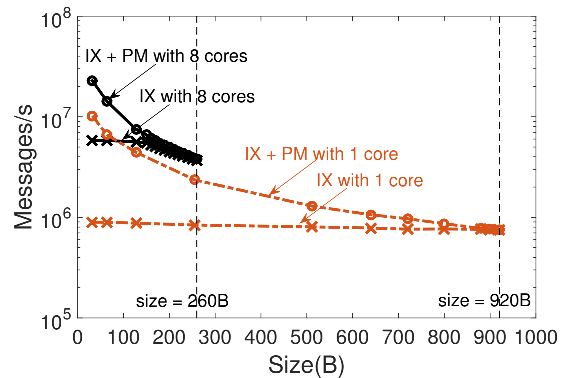


Figure 9: The throughput with different payload sizes for IX and IX + PostMan (PostMan enables up to six helper nodes).

Both the Memcached and the Paxos experiments have confirmed the effectiveness of PostMan: for services processing small packets, PostMan can quickly mitigate the long latency caused by unexpected bursty traffic, because it can offload the overhead of packet processing to helpers; data migration, on the other hand, takes much longer to achieve the same goal, and may further increase the latency during data migration when data migration competes for resource with processing clients' requests.

Capabilities of PostMan. To understand in what circumstances can PostMan help to mitigate bursty traffic, we measure how Memcached's and Paxos' tail latency grow with the load and how PostMan changes such trend.

As shown in Figure 6 and Figure 7, both Memcached get experiment and Paxos experiment show the same trend: when the load is low (i.e., lower than 2000K messages/s in Memcached and 500K messages/s in Paxos), using PostMan will actually introduce extra latency, because of the additional processing at the helper; when the load grows, the latency of the original systems will grow as well due to queuing delay and when these systems are close to saturation, their latency jumps, which is what happens when the service experiences bursty traffic. PostMan can offload their overhead of packet processing to helpers and thus can improve their maximum throughput, which reduces their queuing delay in a range of loads: for Memcached get experiments, PostMan can reduce the latency if the load is between 2000K and 6000K messages/s; for Paxos, PostMan can reduce its latency if the load is between 500K and 5000K messages/s. Memcached set experiments do not benefit from PostMan, because as shown in our profiling, its bottleneck is lock contention, which has nothing to do with packet processing. This set of experiments show that PostMan is effective for a wide range of loads, but it does have its limits: that's why it is complementary to data migration, which does not have such limits but requires a longer time to be effective.

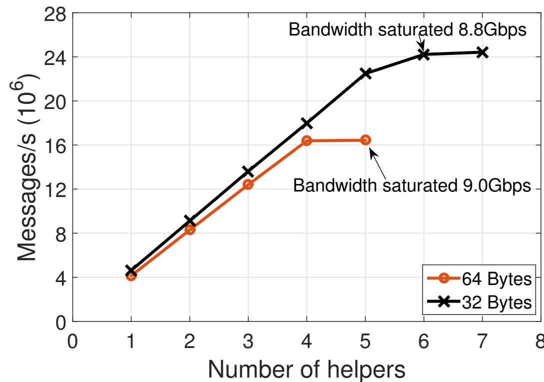


Figure 10: The performance scale linearly when increasing the number of helpers nodes.

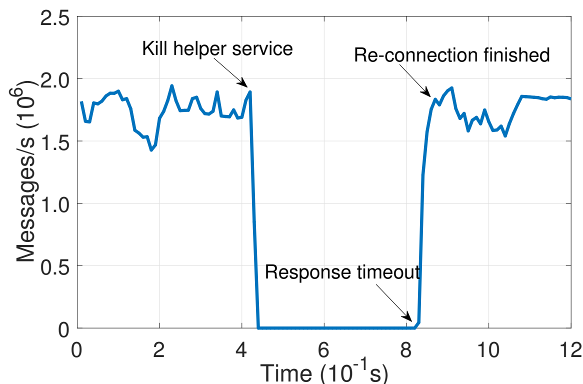


Figure 11: The performance when PostMan recovers the connections by mapping them to another active helper.

6.2 Effects of packet size

The capability of PostMan is affected by the packet size because PostMan's key idea of assembling packets naturally works well with smaller packets. To quantitatively understand how PostMan's capability is affected by this factor, we use the ping-pong microbenchmark to measure the throughput of PostMan, since as shown in Section 6.1, the maximal throughput of PostMan determines the range of loads in which PostMan might be helpful. We compare the throughput of PostMan to those of Linux and IX [7]. Since we fail to run an IX server with more than nine cores, we reduce the number of cores to eight in this set of experiments. Since IX shows it can outperform mTCP [24], another popular network stack, we do not further compare PostMan with mTCP.

Figure 8 shows the throughput of Linux under different packet sizes, with and without PostMan. As shown in this figure, when the server can utilize 8 cores for packet processing, PostMan can improve throughput when the payload size is less than 400 bytes. However, for CPU-intensive applications, this may not be a fair comparison because PostMan can reduce CPU utilization as well as improving throughput. Therefore, we also show the comparison when the server can utilize only one core for packet processing. In this case, PostMan can improve throughput when payload size is smaller than 1460 bytes.

Figure 9 shows the throughput of IX under different packet sizes, with and without PostMan. It shows a similar trend as the Linux experiment, though the turning points are smaller, 260B and 920B respectively. The benefit of PostMan still exists although becomes smaller than that in the Linux experiment. This is because IX, with an optimized networking stack, pays less overhead per packet compared to Linux.

Comparing Figure 8 and Figure 9, one can see that Linux+Postman can even outperform IX when the packet size is small, despite the fact that the former approach does not require installing a new OS on all servers.

6.3 Performance of helper nodes

So far we have measured the performance gain at the server side. A natural question is how much resource we need to pay at the helper side to achieve such performance gain. To answer this question, we measure how much throughput a single helper can provide and whether PostMan scales with the number of helpers.

In this set of experiment, we use IX as the server side to ensure the server will not become the bottleneck. As Figure 10 shows, a single helper node can process about 9.6 million small messages (assembling 4.8 million requests and disassembling 4.8 million responses) per second. When increasing the number of helper nodes, the overall throughput of PostMan scales almost linearly, till the helpee's bandwidth is saturated.

Such results have demonstrated that PostMan does need a number of helper nodes to improve the throughput at the server side: this is as expected because PostMan offloads overhead instead of reducing overhead. Therefore, we expect a small-scale deployment of PostMan to help a few heavily loaded servers.

6.4 Fault tolerance

To test whether PostMan can tolerate failures of helper nodes, we set up a simple scenario, in which two active helper nodes are connected to the server. A Linux client running on PostMan client library is performing request-reply communication to the server, with 10 threads and 1000 connections in total. To examine the failure recovery of helper nodes, we first let all clients connect to one helper, record the throughput (measured every 100ms) of the clients and manually kill the connected helper node. As shown in Figure 11, the library waits until a timeout on receiving the reply message, and then reconnects to the other helper node. The re-connection takes about

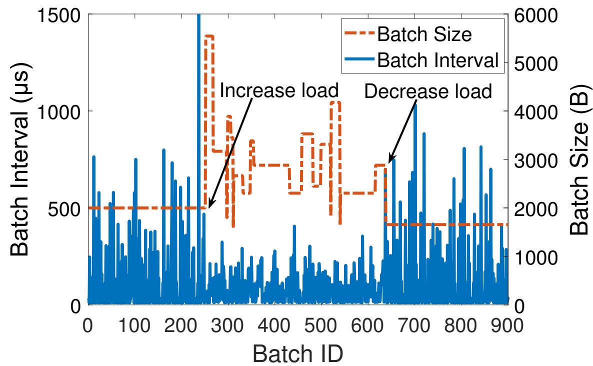


Figure 12: Adaptively changing batch size and interval.

0.4s to recover all 1000 connections. After that, the message exchange reaches the previous rate.

Note that PostMan's correctness does not rely on the correctness of timeout: even if the timeout is inaccurate (i.e., a timeout is triggered when the previous helper node is still alive), PostMan can still guarantee all its properties because clients and servers will close the old connection and exchange necessary information when establishing a new connection (Section 4.2). This means in practice, the developers can use a shorter timeout to improve availability. In this experiment, we simply use an arbitrary 1-second timeout to show that PostMan can function correctly despite failures.

6.5 Adaptive batching

To test the effectiveness of our adaptive batching algorithm, we change the load of our system and see how PostMan reacts. As shown in Figure 12, with the load increasing, the batch interval decreases, implying that the helper nodes batch packets more frequently. Although the size variation seems noisy, the batch sizes are mostly larger than those with low load. When the load decreases, the helper nodes can reduce the batch size and increase the batch interval. As a result, we can conclude that the batch size and batch interval are well adapted to fluctuating load.

7 Conclusion

In this paper, we present PostMan, a distributed service to mitigate load imbalance caused by bursty traffic, by offloading the overhead of packet processing from heavily-loaded servers and reducing data redundancy in packet headers. By batching small packets remotely and on demand, PostMan can utilize multiple nodes to help a heavily-loaded server when bursty traffic occurs and can minimize the overhead when there is no such bursty traffic. Experiments with Memcached and Paxos show that, compared to data migration, PostMan can quickly mitigate the extra latency caused by bursty traffic.

8 Acknowledgment

We thank our shepherd Boris Grot and other anonymous reviewers for their insightful comments. This work was supported in part by the NSFC under Grant 61761136014 (and 392046569 of NSFC-DFG) and 61722206 and 61520106005, in part by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, in part by the Fundamental Research Funds for the Central Universities under Grant 2017KFKJXX009 and 3004210116, in part by the National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals, and in part by the NSF grant CNS-1566403.

References

[1] Dpdk (data plane development kit). https://www.dpdk.org/.

[2] Memcached. http://memcached.org.

[3] Receive side scaling on intel network adapters. https://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000006703.html.

[4] Apache HBASE. http://hbase.apache.org/.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of SIGMETRICS*, 2012.

[6] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of CIDR*, 2011.

[7] 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 *Proceedings of OSDI*, 2014.

[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 *Proceedings of OSDI*, 2014.

[9] 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 ATC*, 2013.

[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of SOSP*, 2011.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI*, 2006.

[12] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of EuroSys*, 2016.

[13] CloudLab. https://cloudlab.us.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, 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 *Proceedings of OSDI*, 2012.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[16] 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 SOSP*, 2007.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of SOSP*, 2003.

[18] Glastonbury ticket website crashes. https://www.theguardian.com/music/2016/oct/09/glastonbury-ticket-website-crashes, 2016.

[19] Chamara Gunaratne, Ken Christensen, and Bruce Nordman. Managing energy consumption costs in desktop pcs and lan switches with proxying, split tcp connections, and scaling of link speed. *International Journal of Network Management*, 15(5):297–310, 2005.

[20] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. Elmem: Towards an elastic memcached system. In *Proceedings of ICDCS*, 2018.

[21] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of ACM symposium on Cloud computing*, 2010.

[22] N. Islam, W. Rahman, X. Lu, and D. Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of ICS*, 2016.

[23] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of NSDI*, 2014.

[24] 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 *Proceedings of NSDI*, 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of ATC*, 2016.

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

[27] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance rdma systems. In *Proceedings of ATC*, 2016.

[28] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: high-performance in-memory key-value store with programmable nic. In *Proceedings of SOSP*, 2017.

[30] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. Dhl: Enabling flexible software network functions with fpga acceleration. In *Proceedings of ICDCS*, 2018.

[31] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[32] X. Lu, D. Shankar, S. Gugnani, and D. Panda. High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads. In *Proceedings of IEEE International Conference on Big Data*, 2016.

[33] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of OSDI*, 2004.

[34] Macy's Web Site Buckles Under Heavy Traffic on Black Friday. http://fortune.com/2016/11/25/macys-black-traffic/, 2016.

[35] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of CoNEXT*, 2014.

[36] Introduction to Parallel I/O. https://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf.

[37] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of OSDI*, 2012.

[38] 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 *Proceedings of NSDI*, 2013.

[39] Yipei Niu, Fangming Liu, Xincai Fei, and Bo Li. Handling flash deals with soft guarantee in hybrid cloud. In *Proceedings of INFOCOM*, 2017.

[40] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *Proceedings of INFOCOM*, 2015.

[41] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP*, 2011.

[42] Abhinav Pathak, Y. Angela Wang, Cheng Huang, Albert Greenberg, Y. Charlie Hu, Randy Kern, Jin Li, and Keith W. Ross. Measuring and evaluating tcp splitting for cloud services. In *Proceedings of International Conference on Passive and Active Measurement*, 2010.

[43] 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 *Proceedings of OSDI*, 2014.

[44] W. Rahman, X. Lu, N. Islam, R. Rajachandrasekar, and D. Panda. High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA. In *Proceedings of IPDPS*, 2015.

[45] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[46] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of USENIX Security Symposium*, 2012.

[47] Marcel-Catalin Rosu and Daniela Rosu. An evaluation of tcp splice benefits in web proxy servers. In *Proceedings of WWW*, 2002.

[48] D. Shankar, X. Lu, N. Islam, W. Rahman, and D. Panda. High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits. In *Proceedings of IPDPS*, 2016.

[49] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of OSDI*, 2016.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of MSST*, 2010.

[51] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of SOPS*, 2015.

R2P2: Making RPCs first-class datacenter citizens

Marios Kogias George Prekas Adrien Ghosn Jonas Fietz Edouard Bugnion

EPFL, Switzerland

Abstract

Remote Procedure Calls are widely used to connect datacenter applications with strict tail-latency service level objectives in the scale of µs. Existing solutions utilize streaming or datagram-based transport protocols for RPCs that impose overheads and limit the design flexibility. Our work exposes the RPC abstraction to the endpoints and the network, making RPCs first-class datacenter citizens and allowing for in-network RPC scheduling.

We propose R2P2, a UDP-based transport protocol specifically designed for RPCs inside a datacenter. R2P2 exposes pairs of requests and responses and allows efficient and scalable RPC routing by separating the RPC target selection from request and reply streaming. Leveraging R2P2, we implement a novel *join-bounded-shortest-queue (JBSQ)* RPC load balancing policy, which lowers tail latency by centralizing pending RPCs in the router and ensures that requests are only routed to servers with a bounded number of outstanding requests. The R2P2 router logic can be implemented either in a software middlebox or within a P4 switch ASIC pipeline.

Our evaluation, using a range of microbenchmarks, shows that the protocol is suitable for µs-scale RPCs and that its tail latency outperforms both random selection and classic `HTTP` reverse proxies. The P4-based implementation of R2P2 on a Tofino ASIC adds less than 1µs of latency whereas the software middlebox implementation adds 5µs latency and requires only two CPU cores to route RPCs at 10 Gbps line-rate. R2P2 improves the tail latency of web index searching on a cluster of 16 workers operating at 50% of capacity by 5.7× over NGINX. R2P2 improves the throughput of the Redis key-value store on a 4-node cluster with master/slave replication for a tail-latency service-level objective of 200µs by more than 4.8× vs. vanilla Redis.

1 Introduction

Web-scale online data-intensive applications such as search, e-commerce, and social applications rely on the scale-out architectures of modern, warehouse-scale datacenters to meet service-level objectives (SLO) [7, 17]. In such deployments, a single application can comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers and connected by commodity Ethernet switches. The typical pattern for web-scale applications distributes the critical data (*e.g.,* the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [26, 85, 87–89], NoSQL databases [62, 78], key-value stores [22,54,59,67,93], or specialized graph stores [14]. Consequently, online data-intensive (OLDI) applications are deployed as 2-tier applications with root servers handling end-user queries and leaf servers holding replicated, sharded data [8, 58]. This leads to a high fan-in, high fan-out connection graph between the tiers of an application that internally communicates using RPCs [11]. Each client must (a) fan-out an RPC to the different shards and (b) within each shard, choose a server from among the replica set. Moreover, each individual task can require from only few microseconds (µs) of user-level execution time for simple key-value requests [54] to a handful of milliseconds for search applications [35].

To communicate between the tiers, applications most commonly layer RPCs on top of TCP, either through RPC frameworks (*e.g.,* gRPC [31] and Thrift [86]) or through application-specific protocols (*e.g.,* Memcached [59]). This leads to a mismatch between TCP, which is a byte-oriented, streaming transport protocol, and message-oriented RPCs. This mismatch introduces several challenges, one of which is RPC load distribution. In one approach, root nodes randomly select leaves via direct connections or L4-load balancing. This approach leads to high fan-in, high fan-out communication patterns, load-imbalance and head-of-line blocking. The second approach uses a L7 load balancer or reverse proxy [1, 16, 25] to select among replicas on a per request basis, *e.g.,* using a Round-Robin or Join-Shortest-Queue (JSQ) algorithm. While such load balancing policies improve upon random selection, they do not eliminate head-of-line blocking. Furthermore, the load balancer can become a scalability bottleneck.

This work proposes a new communication abstraction for datacenter applications that exposes RPCs as first-class citi-

zens of the datacenter not only at the client and server endpoints, but also in the network. Endpoints have direct control over RPC semantics, do not suffer from head-of-line blocking because of connection multiplexing, and can limit buffering at the endpoints. The design also enables RPC-level processing capabilities for in-network software or hardware middleboxes, including scheduling, load-balancing, straggler-mitigation, consensus and in-network aggregation.

As a first use case, we show how to use our network protocol to implement efficient, scalable, tail-tolerant, high-throughput routing of RPCs. Our design includes an RPC router that can be implemented efficiently either in software or within a programmable switch ASIC such as P4 [12]. In addition to classic load balancing policies, we support *Join-Bounded-Shortest-Queue* (`JBSQ(n)`), a new RPC scheduling policy that splits queues between the router and the servers, allowing only a bounded number of outstanding requests per server, which significantly improves tail-latency.

We make the following contributions :

- The design of *Request-Response Pair Protocol (R2P2)*, a transport protocol designed for datacenter µs-RPCs that exposes the RPC abstraction to the network and the endpoints, breaks the point-to-point RPC communication assumptions, and separates request selection from message streaming, nearly eliminating head-of-line blocking.
- The implementation of the R2P2 router on a software middlebox that adds only 5µs to end-to-end unloaded latency and is capable of load balancing incoming RPCs at line rate using only 2 cores.
- The implementation of the R2P2 router within a P4-programmable Tofino dataplane ASIC, which eliminates the I/O bottlenecks of a software middlebox and reduces latency overhead to 1µs.
- The implementation of `JBSQ(n)`, a split-queue scheduling policy that utilizes a single in-network queue and bounded server queues and improves tail-latency even for µs-scale tasks.

Our evaluation with microbenchmarks shows that our R2P2 deployment with a `JBSQ(3)` router achieves close to the theoretical optimal throughput for 10µs tasks across different service time distributions for a tail-latency SLO of 150µs and 64 independent workers. Running Lucene++ [56], an open-source websearch library over R2P2, shows that R2P2 outperforms conventional load balancers even for coarser-grain, millisecond-scale tasks. Specifically, R2P2 lowers the 99th percentile latency by 5.7× at 50% system load over NGINX with 16 workers. Finally, running Redis [78], a popular key-value store with built-in master-slave replication, over R2P2 demonstrates an increase of 4.8×– 5.6× in throughput vs. vanilla Redis (over TCP) at a 200µs tail-latency SLO for different `read:write` ratios. The Redis improvements are due to the cumulative benefits of a leaner protocol, kernel bypass, and scheduling improvements.

The paper is organized as follows: §2 provides the necessary background. §3 describes the R2P2 protocol and §4 its implementation. §5 is the experimental evaluation of R2P2. We discuss related work in §6 and conclude in §7. The R2P2 source code is available at `https://github.com/epfl-dcsl/r2p2`.

2 Background

2.1 Datacenter RPCs

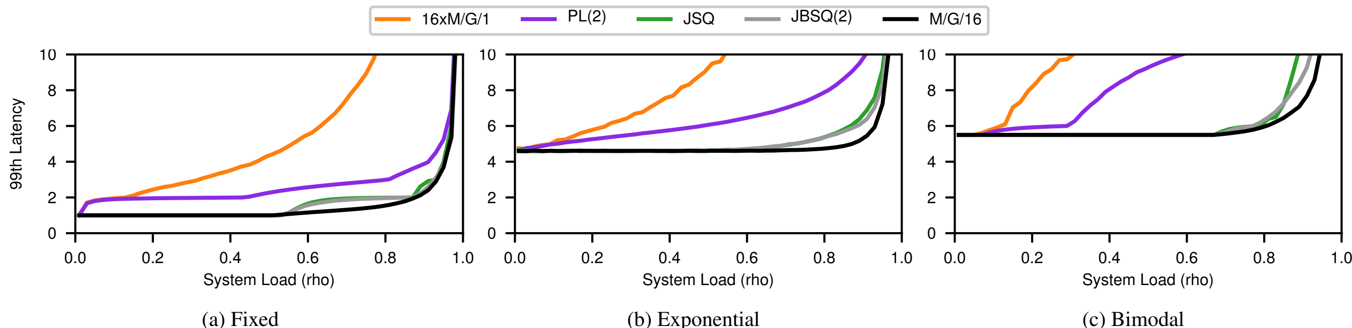
TCP has emerged as the main transport protocol for latency-sensitive, intra-datacenter RPCs running on commodity hardware, as its reliable stream semantics provide a convenient abstraction to build upon. Such use is quite a deviation from the original design of a wide-area, connection-oriented protocol for both interactive (*e.g.,* `telnet`) and file transfer applications. TCP's generality comes with a certain cost as RPC workloads usually consist of short flows in each direction. In many cases, the requests and replies are small and can fit in a single packet [5,63]. Although RDMA is an alternative, it has specific hardware requirements and can be cumbersome to program, leading to application-specific solutions [22,42,43].

Overall, the requirements of RPCs differ from the assumptions made by TCP in terms of failure semantics, connection multiplexing, API scalability, and end-point buffering:

RPC semantics: Some datacenter applications choose weak consistency models [18] to lower tail latency. These applications typically decompose the problem into a series of independent, often idempotent, RPCs with no specific ordering guarantees. Requests and responses always come in pairs that are semantically independent from other pairs. Thus, the reliable, in-order stream provided by TCP far stronger than the applications needs and comes with additional network and system overheads.

Connection multiplexing: To amortize the setup cost of TCP flows, RPCs are typically layered on top of persistent connections, and most higher-level protocols support multiple outstanding RPCs on a flow, *e.g.,* `HTTP/2`, memcache, *etc.* Multiplexing different RPCs on the same flow implies ordering the requests that share a socket, even though the individual RPCs are semantically independent. This ordering limits scheduling choices and can lead to Head-of-Line-Blocking (HOL). HOL appears when fast requests are stuck behind a slower request and when a single packet drop affects multiple pending requests.

Connection scalability: The high fan-in, high fan-out patterns of datacenter applications lead to large number of connections and push commodity operating systems beyond their



(a) Fixed (b) Exponential (c) Bimodal

Figure 1: Simulation results for the 99th percentile latency across 3 service time distributions with $\bar{S} = 1$

efficiency point. Recent work has addressed the issue either by deviating from the POSIX socket interface while maintaining TCP as the transport [9] or by developing custom protocols, *e.g.,* to deploy memcached on a combination of connection-less UDP for RPC `get` and router proxy for RPC `set` [67].

Endpoint bufferbloat: Prior work has addressed network-specific issues of congestion management and reliability within the network [2, 3]. Unfortunately, the use of TCP via the POSIX socket API leads to buffering in both endpoints over which applications have little control or visibility [45]. Applications willing to trade-off harvest vs. yield [29] would ideally never issue RPCs with no chance of returning by the deadline due to buffering in the network stack.

2.2 Load balancing

The problem of spreading out load extends to load balancing across servers within a distributed, scale-out environment. Load balancers encapsulate a set of servers behind a single virtual IP address and improve the availability and capacity of applications. Load balancing decisions, however, can severely affect throughput and tail-latency; thus, a significant amount of infrastructure is dedicated to load balancing [23, 60]. Load balancers can be implemented either in software [23, 64, 69] or in hardware [1, 16, 25, 60] and fall into two broad categories: (1) Layer-4 ("network") load balancers that use the 5-tuple information of the TCP or UDP flow to select a destination server. The assignment is static and independent of the load; (2) Layer-7 ("application") load balancers come in the form of `HTTP` reverse proxies as well as protocol-specific routers implemented in software middleboxes [67] or SDN switches [13, 15]. These load balancers terminate the client TCP connections, use dynamic policies to select a target, and reissue the request to the server on a different connection.

Layer-7 load balancers support many policies to decide the eventual RPC target, including random, power-of-two [61], round-robin, Join-Shortest-Queue (JSQ), and Join-Idle-Queue (JIQ) [55]. Layer-7 load balancers are ubiquitous at the web tier and can theoretically mitigate tail-latency better, due to their dynamic policies. However, they are less commonly deployed within tiers of applications to support µs-scale RPCs. The reasons for this are (i) the increased latency due to the extra hop (ii) the scalability issues introduced when all requests and responses flow through a proxy.

2.3 As a queuing theory problem

In this section, we approach the problem of RPC load balancing from a theoretical point of view by abstracting away system aspects using basic queuing theory. We show the benefits of request-level load balancing over random-selection among distributed queues (which is equivalent to L4 load balancing) in improving tail-latency, and we evaluate different request-level load balancing policies.

Fortunately, the theoretical answers are clear: single-queue, multi-worker models (*i.e.,* `M/G/k` according to Kendall's notation) perform better than distributed multi-queue models (*i.e.,* `k×M/G/1`, with one queue per worker) because they are work-conserving and guarantee that requests are processed in order [49, 90].

Between those two extremes, there are other models that improve upon random selection and are practically implementable through L7 load balancing. Power-of-two [61] (`PL(2)`), or similar schemes, are still in the realm of randomized load balancing, but perform better than a blind random selection. `JSQ` performs close to a single queue model for low-variability service times [55].

We define Join-Bounded-Shortest-Queue `JBSQ(n)` as a policy that splits queues between a centralized component with an unbounded queue and distributed bounded queues of maximum depth `n` for each worker (including the task currently processed). The single-queue model is equivalent to `JBSQ(1)` whereas `JSQ` is equivalent to `JBSQ(∞)`.

Figure 1 quantifies the tail-latency benefit, at the 99th percentile, for these queuing models observed in a discrete event simulation. We evaluate a configuration with a Poisson arrival process, $k = 16$ workers, and three well-known distributions with the same service time $\bar{S} = 1$. These distributions are: deterministic, exponential and bimodal-1 (where 90% of re-

quests execute in .5 and 10% in 5.5 units) [55].

From the simulation results, we conclude that: (1) there is a dramatic gap in performance between the random, multi-queue model and the single-queue approach, which is optimal among FCFS queuing systems. (There is no universally optimal scheduling strategy for tail-latency [90].) (2) `PL(2)` improves upon random selection, but these benefits diminish as service time variability increases. `JSQ` performs close to the optimal for low service time variability. (3) `JBSQ(2)`, while it deviates from the single queue model, outperforms `JSQ` under high load as the service time variability increases.

These results are purely theoretical and in particular assume perfect global knowledge by the scheduler or load balancer. This global view would be the result of communication between the workers and the load balancer in a real deployment. Any practical system must consider I/O bottlenecks and additional scheduling delays because of this communication. In this paper, we make the claim that `JBSQ(n)` can be implemented in a practical system and can deliver maximal throughput with small values of n even for µs-scale tasks, thus minimizing tail latency and head-of-line blocking.

3 R2P2: A transport protocol for RPCs

We propose R2P2 (*Request-Response Pair Protocol*), a UDP-based transport protocol specifically targeting latency-critical RPCs within a distributed infrastructure, *i.e.,* a datacenter. R2P2 exposes the RPC abstraction to the network, thus allowing for efficient in-network request-level load balancing.

R2P2 is a connectionless transport protocol capable of supporting higher-level protocols such as `HTTP` without protocol-level modifications. Unlike traditional multiplexing of the RPC onto a reliable byte-oriented connection, R2P2 is an inherently request/reply-oriented protocol that maintains no state across requests. The R2P2 request-response pair is initiated by the client and is uniquely identified by a triplet of $< src_IP, src_port, req_id >$. This design choice decouples the request destination (set by the client) from the actual server that will reply, thus breaking the point-to-point RPC communication semantics and enabling the implementation of any request-level load balancing policy.

Figure 2 describes the interactions and the packets exchanged in sending and receiving an RPC within a distributed infrastructure that uses a request router to load balance requests across the servers. We illustrate the general case of a multi-packet request and a multi-packet response.

1. A `REQ0` message opens the RPC interaction, uniquely defined by the combination of source IP, UDP port, and an RPC sequence number. The datagram may contain the beginning of the RPC request itself.

2. The router identifies a suitable target server and directs the message to it. If there is no available server, requests can temporarily queue up in the router.

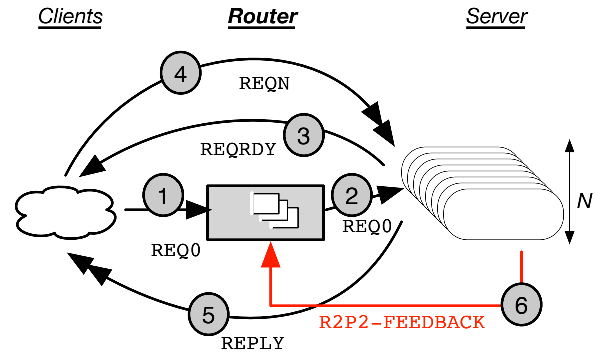


Figure 2: The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams.

3. If the RPC request exceeds the size of data in the `REQ0` payload, then the server uses a `REQready` message to inform the client that it has been selected and that it will process the request.

4. Following (3), the client directly sends the remainder of the request as `REQn` messages.

5. The server replies directly to the client with a stream of `REPLY` messages.

6. The servers send `R2P2-FEEDBACK` messages to the router to signal idleness, availability, or health, depending on the load balancing policies.

We note a few obvious consequences and benefits of the design: (i) Given that an RPC is identified by the triplet, responses can arrive from a different machine than the original destination. Responses are sent directly to the client, bypassing the router; (ii) there is no head-of-line blocking resulting from multiplexing RPCs on a socket, since there are no sockets and each request-response pair is treated independently; (iii) there are no ordering guarantees across RPCs; (iv) the protocol is suited for both short and long RPCs. By avoiding the router for `REQn` message and replies, the router capacity is only limited by its hardware packet processing rate, not by the overall amount of size of the messages.

Unlike protocols that blindly provide reliable message delivery, R2P2 exposes failures and delays to the application. R2P2 follows the end-to-end argument in systems design [80]. A client application initiates a request-response pair and determines the failure policy of each RPC according to its specific needs and SLOs. By propagating failures to the application, the developer is free to choose between *at-least-once* and *at-most-once* semantics by re-issuing the same request that failed. Unlike TCP, failures affect only the RPC in question, not other requests. This is useful in cases with fan-out replicated

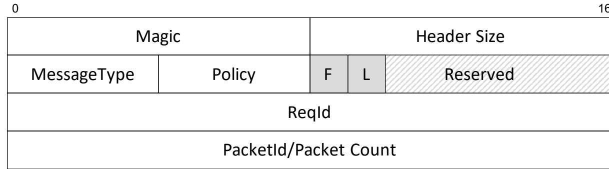


Figure 3: R2P2 Header Format

Message	Description
REQUEST	A message carrying an RPC request
REPLY	A message carrying an RPC reply
REQRDY	Sent by the server to the client to ack the REQ0 of a multi-packet request
R2P2-FEEDBACK	Sent by the server to the router
DROP	Sent by the router or the server to a client to explicitly drop a request
SACK	Sent by the client or the server to ask for missing packets in a request or reply

Table 1: The R2P2 message types

requests, where R2P2 can provide system support for the implementation of tail-mitigation techniques, such as hedged requests [17].

While novel in the context of µs-scale, in-memory computing, the connection "pair" is similar in spirit to the "exchange" that is the core of the SCSI/Fibre Channel protocol (FCP [27]). For example, a single-packet-request-multi-packet-response RPC over R2P2 would be similar to SCSI `read` within a single fibre channel exchange. Equivalently, an R2P2 multi-packet-request-single-packet-response would be similar to a SCSI `write`.

3.1 Transport considerations

Figure 3 describes a proposed R2P2 header, while Table 1 includes the different R2P2 messages. All R2P2 messages are UDP datagrams. R2P2 supports a 16-bit request id whose scope is local to the `(src_ip, src_port)` pair. As such, each client (`(src_ip, src_port)` pair) can have up to 65536 outstanding RPCs, well beyond any practical limitations. The R2P2 header also includes a 16-bit packet id meaning that each R2P2 message can consist of up to 65536 MTU-sized packets. The above two fields can be extended, if necessary, without changing the protocol semantics. Currently R2P2 uses two flags (`F, L`) to denote the first and last packet of a request.

Finally, the R2P2 header contains a `Policy` field, which allows client applications to directly specify certain policies to the router, or any other intermediate middlebox, for this specific RPC. Currently, the only implemented policies are `unrestricted`, which allows the router to direct `REQ0` packet to any worker in the set, and `sticky`, which forces the router to direct the message to the master worker among the set. This mechanism is central to our implementation of a tail-tolerant Redis, based on a master/slave architecture. It is used to direct writes to the master, but balances reads according to the load balancing policy. Additional policies, *e.g.,* session-stickiness, or policies implementing different consistency models, can be implemented in R2P2 middleboxes and will be identified by this header field, thus showcasing the benefits of R2P2's in-network RPC awareness.

Deployment assumptions: We assume that R2P2 is deployed within a datacenter, *i.e.,* the clients, router and servers are connected by a high-bandwidth, low-latency Ethernet fabric. We make no assumptions about the core network that can depend either on ECMP flow hashing or packet spraying [30, 32, 63]. R2P2 tolerates packet reordering within the same message and reconstructs the message at the end-point. By design, though, there is no ordering guarantee across RPCs, even if they are sent by the same client.

Timer management: Given that the assumed deployment model allows for packet reordering, packet loss detection depends on timers. There is one retransmission timeout `RTO` timer used for multi-packet requests or responses. It is in the order of milliseconds and triggers the transmission of a `SACK` message request for the missing packets. Servers garbage collect RPCs with failed multi-packet requests or multi-packet replies after a few `RTO`s. On the client side there is a timer set by the client application when sending the request. This timer is disarmed when the whole reply is received, and can be as aggressive as the application SLO. Based on this timer applications can implement tail-mitigation techniques [17] or early drop requests based on their importance.

Congestion management: R2P2 focuses on reducing queuing on the server side; we do not make any explicit contribution in congestion control. Instead, R2P2 can utilize existing solutions for congestion control, including (1) Homa [63], whose message semantics easily map to R2P2's request-response semantics and (2) ECN-based schemes such as DCTCP [2] and DCQCN [94]. Congestion control will be necessary only for multi-packet requests and replies (`REQN` and `REPLY`), and is independent of the interactions described in Fig 2.

Flow Control: R2P2 implements two levels of flow control, one between the client and the middlebox and one between the middlebox and the servers. R2P2 middleboxes can drop individual requests, either randomly or based on certain priority policies, if they become congested, without affecting other requests, thus implementing the first level of flow control. Based on the functionality and the policy, the middlebox is in charge of implementing the second level of flow control to the servers. In the `JBSQ` case, `JBSQ` limits the number of

outstanding requests on each server, thus servers can not be overwhelmed.

3.2 API

R2P2 exposes a non-POSIX API specifically designed for RPC workloads. Making RPCs first class citizens and exposing the request-response abstraction through the networking stack significantly simplifies writing client-server applications. Application code that traditionally implements the RPC logic on top of a byte stream abstraction is now part of the R2P2 layer of the networking stack.

Table 2 summarizes the corresponding application calls and callbacks for the client and server application. The API has an asynchronous design that allows applications to easily send and receive independent RPCs. When calling `r2p2_send_req` the client application sets the timer timeout and callback functions independently for each RPC request. The client and server applications are notified only when the entire response or request messages have arrived through the `req_success` and `req_recv` callbacks, equivalently.

3.3 `JBSQ` router design considerations

R2P2 exposes the request-response abstraction to the network as a first-class citizen. It is expected that a software or hardware middlebox will manipulate client requests to implement a certain policy, *e.g.,* scheduling, load balancing, admission control, or even application logic, *e.g.,* routing requests to the right server in a distributed hash table. In this section, we discuss the design choices regarding an R2P2 request router implementing the `JBSQ` scheduling policy. Similar ideas can be applied to other middleboxes with alternative functionality.

The choice of `JBSQ`: As seen in § 2.3 `JSQ` and `JBSQ` perform closer to the optimal single queue model. `JBSQ` though offers several practical benefits over `JSQ`. It implements router-servers flow control and can be implemented within a Tofino ASIC. JSQ requires finding the minimum among a number of values, which is hard to implement in a hardware dataplane. Also, JBSQ achieves better latency under high load and service time dispersion. That is because JSQ uses the queue size as a proxy for queuing time, which can be misleading in the presence of service-time dispersion.

Application Calls	
Type	**Description**
r2p2_poll	Poll for incoming req/resp
r2p2_send_req	Send a request
r2p2_send_response	Send a response
r2p2_message_done	Deallocate a request or response

Callbacks	
Type	**Description**
req_recv	Received a new request
req_success	Request was successful
req_timeout	Timer expired
req_error	Error condition

Table 2: The `r2p2-lib` API

R2P2-FEEDBACK messages: To implement the `JBSQ(n)` policy we leverage the `R2P2-FEEDBACK` messages provided by the R2P2 specification. These messages, sent by the servers back to the router after completing the service of a request, specify: (i) The maximum number of outstanding RPCs the server is willing to serve (the "`n`" in `JBSQ(n)`). By sending the current "`n`" in every `R2P2-FEEDBACK` message, servers can dynamically change the number of outstanding requests based on the application SLOs. (ii) The number of requests this server has served including the last request. The router uses this information to track the current number of outstanding requests in the server's bounded queue. This field makes the message itself idempotent and the protocol robust to `R2P2-FEEDBACK` drops.

We note that this approach puts each server in charge of controlling its own lifecycle by sending unsolicited `R2P2-FEEDBACK` messages, *e.g.,* to join a load balancing group, leave it, adjust its bounded queue size based on its idle time, or to periodically signal its idleness.

Direct client request - direct server return: R2P2 implements direct server return (DSR) [34, 65] since the replies do not go through the router. This is a widely-used technique in L4 load balancers with static policies [65]. R2P2 uses DSR while implementing request-level load balancing. In addition, R2P2 implements direct client request, where the router handles only the first packet of a multi-packet request, while the rest is streamed directly to the corresponding server, thus avoiding router IO bottlenecks.

Deployment: A software R2P2 router is deployed as a middlebox and traffic is directed to its IP address. The hardware R2P2 router is also deployed as an IP-addressed middlebox. The same hardware can also be a Top-of-Rack switch serving traffic to servers within the rack, following a "rack-scale" deployment pattern. In such a pattern, the router has full visibility on the RPC traffic to the rack and all packets go through the ToR switch. This could enable simplifications to the packet exchange, *e.g.,* using `R2P2-FEEDBACK` messages only for changing the depth of the bounded queues; the ToR can estimate their current size by tracking the traffic.

Router high availability: The router itself is nearly stateless and a highly-available implementation of the router is relatively trivial. Upon a router failure, only soft state regarding the routing policy is lost, including the current size of the per-worker bounded queue and the queue of pending RPCs. Clients simply failover to the backup router using a virtual IP address and reissue RPCs upon timeout, using the

exact same mechanism used to handle a REQ0 packet loss. Servers reconstruct the relationship with the router with their R2P2-FEEDBACK message to the new router.

Server membership: Servers behind the R2P2 router can fail and new servers can join the load balancing group. R2P2-FEEDBACK messages implicitly confirm to the router that a server is alive. In case of a failure, the lack of R2P2-FEEDBACK messages will prevent the router from sending requests to the failed server, and the bounded nature of JBSQ(n) limits the number of affected RPCs. Similarly, newly-added servers can send R2P2-FEEDBACK messages to the router informing about their availability to serve requests.

The choice of JBSQ(n): The choice of n in JBSQ is crucial. A small n will behave closer to a single-queue model, but will restrict throughput. The rationale behind the choice of n is similar to the Bandwidth Delay Product. On each queue there should be enough outstanding requests so that the server does not stay idle during the server-router communication. For example, for a communication delay of around 15 µs and a fixed service time of 10 µs, n=3 is enough to achieve full throughput. Shorter service times will require higher n values. High service time dispersion and batching on the server will also require higher values than what predicted by the heuristic. Servers can even dynamically adjust the value of n based on their processing rate and minimal idle time between requests.

4 Implementation

We implement (1) r2p2-lib as userspace Linux library on top of either UDP sockets or DPDK [21] (§4.1); (2) the software R2P2 router on top of DPDK (§4.2) and (3) the hardware solution in the $P4_{14}$ programming language [72] to run within a Barefoot Tofino ASIC [6] (§4.3).

4.1 r2p2-lib

The library links into both client and server application code. It exposes the previously described API and abstracts the differences between the Linux socket and the DPDK-based implementations. The current implementation is non-blocking and rpc_poll is typically called in a spin loop. To do so, we depend on epoll for Linux, while for DPDK we implemented a thin ARP, IP, and UDP layer on top of DPDK's polling mode driver, and exposed that to r2p2-lib. Our C implementation of r2p2-lib consists of 1300 SLOC.

R2P2 does not impose any threading model. Given the callback-based design, threads in charge of sending or receiving RPCs operate in a polling loop mode. The library supports symmetric models, where threads are in charge of both network and application processing, by having each thread manage and expose a distinct worker queue through a specific UDP destination port. The DPDK implementation further manages a distinct Tx and Rx queue per thread, and uses Flow Director [36] to steer traffic based on the UDP destination port. In an asymmetric model, a single dispatcher thread links with r2p2-lib, and the other worker threads are in charge of application processing only. This model exposes one worker queue via one UDP destination port.

4.2 Router - software implementation

We implemented a Random, a Round-Robin, a JSQ and a JBSQ(n) policy on the software router. The main implementation requirements for the router are (1) it should add the minimum possible latency overhead, and (2) it should be able to process short REQ0 and R2P2-FEEDBACK messages at line rate. While the router processes only those two types of packets, the order in which it processes them matters. Specifically for JBSQ, the ideal design separates REQ0 from R2P2-FEEDBACK messages into two distinct ingress queues and processes R2P2-FEEDBACKs with higher priority to ensure that the server state information is up-to-date and minimize queuing delays.

Our DPDK implementation uses two different UDP ports, one for each message type, using Flow Director for queue separation. Given the strict priority of control messages and the focus on scalability, we chose a multi-threaded router implementation with split roles for REQ0 threads and R2P2-FEEDBACK threads, with each thread having its own Rx and Tx queues.

JBSQ(n) requires a counter per worker queue that counts the outstanding requests. To minimize cache-coherency traffic, the router maintains two single-writer arrays, one updated on every REQ0 and the other on every R2P2-FEEDBACK, with one entry per worker.

The implementation of the R2P2-FEEDBACK thread is computationally very cheap and embarrassingly scalable. Processing REQ0 messages requires further optimizations to reduce cache-coherency traffic, *e.g.,* maintain the list of known idle workers, cache the current queue sizes, *etc.* Our implementation relies on adaptive bounded batching [9] to amortize the cost of PCIe I/O operations, as well as that of the cache-coherency traffic (the counters are read once per iteration). We limit the batch size to 64 packets.

Finally, we implement a tweak to the JBSQ(n) policy with $n \geq 2$: when no idle workers are present, up to 32 packets are held back for a bounded amount of time on the optimistic view that an R2P2-ACK message may announce the next idle worker. This optimization helps absorb instantaneous congestion and approximate the single-queue semantics in medium load situations.

4.3 P4/Tofino implementation

We built a proof-of-concept P4 implementations of R2P2 router for Tofino [6] using $P4_{14}$ [72]. Similar to the software implementation, the switch only processes REQ0 and

R2P2-FEEDBACK messages and leverages P4 registers to keep soft state. P4 registers are locations in the ASIC SRAM, which can be read and updated from both the control and dataplane.

We focus our description on the implementation of JBSQ(n) for the Tofino dataplane, as the others are trivial in comparison. It consists of 480 lines of P4 source, including header descriptions. Unlike the software implementation that can easily buffer the outstanding REQ0 messages if there is no available server queue, high-performance pipelined architectures, such as Tofino, do not allow buffering in the dataplane. Thus, our P4 logic executes as part of the ingress pipeline of the switch and relies heavily on the ability to recirculate packets through the dataplane via a virtual port. The implementation leverages an additional header that is added to the packet to carry metadata through the various recirculation rounds and is removed before forwarding the packet to the target server.

The logic for REQ0 tries to find a server with $\leq i$ outstanding packets in round i. There is one register instance corresponding to each server, holding the number of outstanding requests. If a suitable server is found, the register value is increased by one, the packet destination is changed to the address of the equivalent server, and the packet is directed to the egress port. We start with $i = 0$ and we increase till $i = n$ from JBSQ(n). When i reaches n and there is still no available server, we keep recirculating the packet without increasing i further. As an optimization to reduce the number of recirculations, the dataplane keeps the i for the last forwarded request and starts from that.

To overcome the Tofino limitation of only being able to compare a limited number of registers in one pass, we also leverage recirculation to inspect the outstanding requests of each bounded queue in each round. Register instances that correspond to different queues are organized in groups that can be checked in one pass. If no available queue is found in the first group, the packet is recirculated (without increasing i) and the second group of queues is checked, *etc.* When a REQ0 arrives, it is initially assigned to a group in a round-robin fashion to further reduce the amount of recirculations.

The logic for R2P2-FEEDBACK decrements the outstanding count for the specific server based on the packet source and consumes the packet without forwarding it.

The use of recirculation has two side-effects: (1) the order of RPCs cannot be guaranteed as one packet may be recirculated while another one is not; (2) the atomicity of the full set of comparisons is not guaranteed as R2P2-FEEDACK packet may be processed while an REQ0 packet is being recirculated. Non-optimal decisions may occur as the result of this race condition.

5 Evaluation

To evaluate the performance and the efficacy of the R2P2 protocol, the two implementations of the router, as well as the trade-offs in using JBSQ(n) over other routing policies, we run a series of synthetic microbenchmarks and two real applications in a distributed setup with multiple servers. The microbenchmarks depend on an RPC service with configurable service time and response size. All our experiments are open-loop [83] and clients generate requests with a Poisson inter-arrival time. We use two baselines and compare them against different configurations for R2P2 with and without the router: (1) vanilla NGINX [66] serving as reverse proxy for HTTP requests; and (2) ZygOS [76], a state-of-the-art work-conserving multicore scheduler. As a load generator we use an early version of Lancet [46].

Our experimental setup consists of cluster of 17 machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The machines are a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). To reduce latency and jitter, we configured the machine that measures latency to direct all UDP packets to the same NIC queue via Flow Director. The Barefoot Tofino ASIC runs within a Edgecore Wedge100BF-32X. The Edgecore is directly connected to the Quanta switch via a 40Gbps link and therefore operates as a 1-port router.

5.1 Router characterization

We use the synthetic RPC service to evaluate the latency overhead of the router, the maximal throughput and the optimal request load balancing policy. We configure a setup of 4 servers with 16 threads (64 independent workers), running the synthetic RPC service over DPDK.

Throughput: We first evaluate the sustainable throughput of the software router. We run a synthetic RPC service with 8-byte requests and we configure the size of the response.

Figure 4 shows the achieved goodput as a function of the response size, and compares a configuration with R2P2 messages handled by a JBSQ load balancing policy, with a NGINX configured as reverse proxy for HTTP messages. For small response sizes, the router is bottlenecked by the router's NIC's packets per second (PPS), or the number of outstanding requests in each queue, n in $JBSQ(n)$. JBSQ(3) was enough to achieve maximum throughput. As the response size increases though, the application goodput converges to $4 \times$ 10GbE, the NIC bottleneck of the 4 servers with payloads as small as 2048. Obviously, this is made possible by the protocol itself, which bypasses the router for all REPLY messages. Note that because R2P2 leverages both Direct Server Return and Direct Client Request, even in cases of large requests the router would not be the bottleneck, unlike traditional L4 DSR-enabled load balancing. In contrast, the NGINX I/O bottleneck limits goodput to the load balancer's 10Gbps NIC.

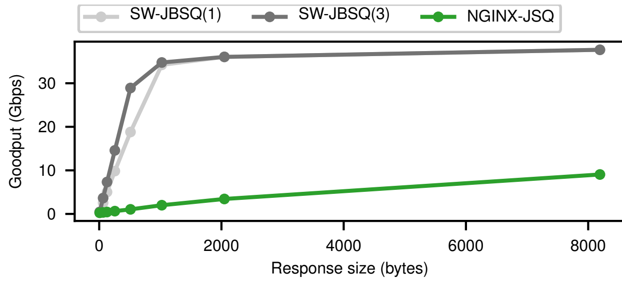


Figure 4: Achieved Goodput as a function of the response size for the JBSQ policy on the software router managing 4 servers connected with 10GbE NICs compared to NGINX configured as HTTP reverse proxy loadbalancing the same 4 servers using a JSQ policy.

Latency overheads and saturation: Figure 5 uses a zero-cost ("echo") RPC service with 8-byte requests and responses, to measure the 99th percentile tail latency as a function of the load for the software middlebox and the Tofino router with the JBSQ policy. As a baseline, we use a DIRECT configuration where clients bypass the router and send requests directly to the servers after a random choice. The figure shows that the latency added by the router is 5µs for the software middlebox and 1µs for the Tofino solution. The software latency is consistent with the characteristics of one-way forwarding performance of the Intel x520 chipset using DPDK. The hardware latency is consistent with the behavior of an ASIC solution that processes and rewrites packet headers in the dataplane. Figure 5 also shows the point of saturation, which corresponds to 7 MRPS for the software middlebox. Given that for every request forwarded the router receives one R2P2-FEEDBACK message, the router handles more than 14M PPS, which is the hardware limit. We were unable to characterize the maximal per-port capability of the Tofino ASIC running the R2P2 logic beyond >8 MRPPS with tiny requests and replies, simply for lack of available client machines. We also observe that the hardware implementation, as expected, requires a smaller n for JBSQ(n). In the figure we show the smallest value of n that achieved maximum throughput.

Comparison of scheduling policies: Figure 6 uses a synthetic $\bar{S} = 25\mu s$ workload to evaluate the different request load balancing policies, implemented on the software router. We evaluate the following policies: DIRECT, where clients bypass the router by making a random server selection, RANDOM where clients talk to the router and the router makes a random selection among the servers, RR where the router selects a target server in a round-robin manner, SW-JBSQ(n) which is the software implementation for the bounded shortest queue with n outstanding requests, and JSQ which is the R2P2 router's implementation of the join-shortest-queue policy. We also compare R2P2 with using NGINX as an HTTP reverse proxy implementing a JSQ policy, which is a vanilla, widely-used

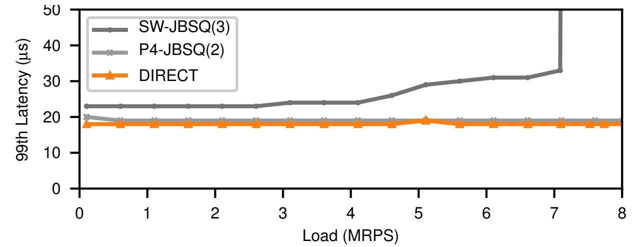


Figure 5: Tail-latency as a function of throughput for a zero service time synthetic RPC service for the software (SW-JBSQ) and the Tofino (P4-JBSQ) implementation of JBSQ compared to DIRECT. In DIRECT clients bypass the router and talk directly to servers by making a random server choice.

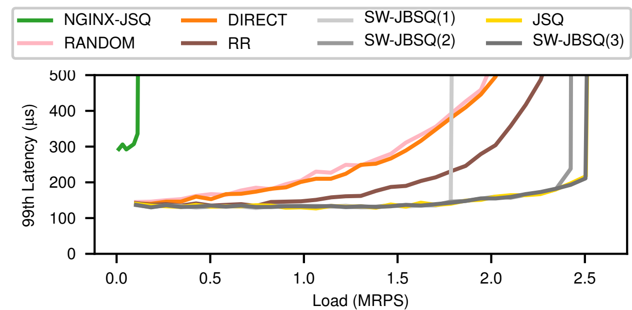


Figure 6: Evaluation of different load balancing policies for an exponential service time workload with $\bar{S} = 25\mu s$.

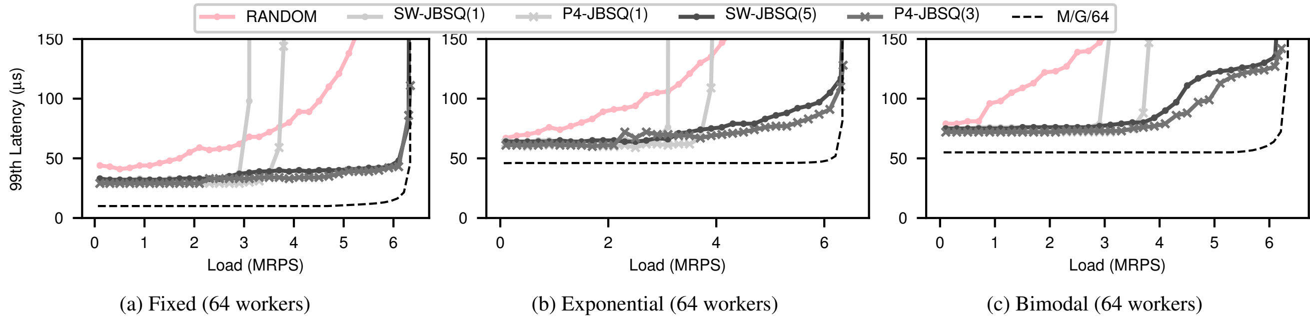
deployment for request-level load balancing.

We make the following observations: (i) NGINX overheads prevent throughput scalability; (ii) DIRECT and RAND configurations perform similarly for R2P2, which is the result of a random choice (in the client or the router equivalently); (iii) RR performs better than random choice, but worse than JBSQ, given the service time dispersion; (iv) JBSQ($n \geq 3$) achieves maximum throughput. Given that the communication time between the server and the router is $\sim 15\mu s$ and the exponential service time dispersion, this is on par with our analysis in § 3.3. (v) JSQ performs similarly to JBSQ(3) for this service time.

5.2 Synthetic Time Microbenchmarks

Figure 7 evaluates JBSQ(n) performance with an aggressive $\bar{S} = 10\mu s$ mean service time and three different service time distributions: Fixed, Exponential and Bimodal where 10% of the request are 10x slower than the rest [55]. We present results for both the software and Tofino implementation, for JBSQ(1) and the optimal n choice for each configuration. Requests and the responses are 8 bytes. We observe:

- For all experiments, all JBSQ(n) variants approximate the optimal single-queue approach ($M/G/64$) until the saturation point for JBSQ(1).



(a) Fixed (64 workers) (b) Exponential (64 workers) (c) Bimodal (64 workers)

Figure 7: Synthetic Service Time Microbenchmarks. Service time $\bar{S} = 10\mu s$.

- Beyond the saturation point of `JBSQ(1)`, an increase in the tail latency as the system configuration trades off higher throughput (*i.e.,* `JBSQ(`$n > 1$`)`) against the use of a theoretically-optimal approach.

- A comparison between the software and hardware implementation shows that more outstanding requests are required for the software implementation; this is because the communication latency between the server and the hardware router is $\sim 5\mu s$ faster.

- `JBSQ` achieves the optimal performance, as predicted by the `M/G/64` model, both for the software and the hardware implementation within the 150µs SLO.

- Reducing `n` can have a considerable impact on tail-latency especially in cases with high service time dispersion, as it can be seen in Figure 7c (`SW-JBSQ(5)` vs. `P4-JBSQ(3)`)

5.3 Multi-packet Requests Microbenchmark

R2P2 implements the following logic in splitting requests to packets. If the request fits in a single packet, the whole request payload is transferred with `REQ0`. In the case of a multi-packet request, `REQ0` is a 64-byte packet, carrying only the first part of the request and the rest of the payload is transferred with the `REQN` packets directly to the server. This way the router does not become a throughput bottleneck in the case of large requests, while the extra round-trip is avoided in the case of small requests.

To evaluate the extra round-trip that R2P2 introduces in the case of multi-packet requests with the distinction between `REQ0` and `REQN`, we ran a synthetic microbenchmark with larger requests. Based on the above logic, a 1464-byte request is the biggest request that fits in a single packet given the size of protocol headers. Equivalently, a 1465-byte request is the smallest request that requires 2 packets, and consequently an extra round-trip. We run the synthetic service time RPC server with the bimodal service time distribution of $\bar{S} = 10$ and the 2 different request sizes. We compare the `DIRECT` deployment with one using the router with the `JBSQ` policy.

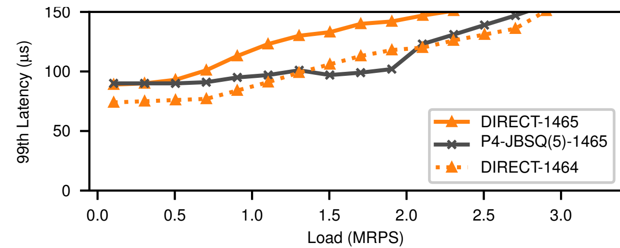


Figure 8: Bimodal service time with $\bar{S} = 10\mu s$ and 64 workers with single and multi-packet requests. `DIRECT-1464` corresponds to an 1-packet request, while `DIRECT-1465` and `P4-JBSQ(5)` correspond to 2-packet requests.

Figure 8 summarizes the result of the experiment. We observe that there is a fixed gap of around 15µs between `DIRECT-1464` and `DIRECT-1465` curves that corresponds to the extra round-trip between the client and the server. We, also, run the multi-packet request scenario while using the P4 router with the `JBSQ` policy. We show that despite the extra round-trip, the intermediate hop, and the increased number of packets to process, the 99th percentile latency is close to the single-packet scenario in the `DIRECT` case, which justifies our design decision to pay an extra round-trip to achieve better scheduling.

5.4 Using R2P2 for server work conservation

We now demonstrate how the use of network-based load balancing, *e.g.,* using R2P2, can increase the efficiency of a *single* server scheduling tasks. For this, we compare R2P2 with `JBSQ` with the performance of ZygOS [76], a state-of-the-art system optimized for µs-scale, multicore computing that includes a work-conserving scheduler within a specialized operating system. ZygOS relies on work-stealing across idle cores and makes heavy use of inter-processor interrupts. Both ZygOS and `JBSQ(n)` offer a work-conserving solution to dispatch requests across the multiple cores of a server: Zy-

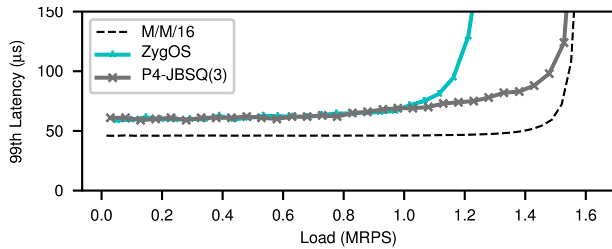


Figure 9: Comparison of R2P2 with the ZygOS [76] work-conserving scheduler: Exponential workload with $\bar{S} = 10\mu s$.

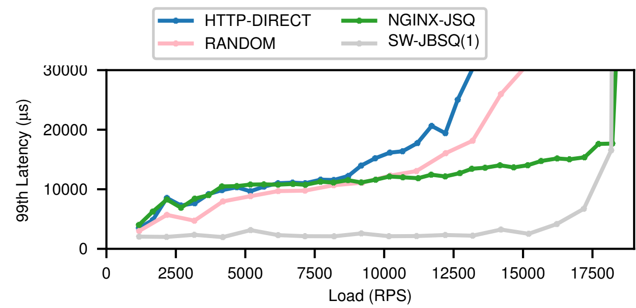


Figure 10: Lucene++ running on 16 16-threaded workers

gOS does it within the server in a protocol-agnostic manner, whereas R2P2 implements the policy in the network.

Figure 9 compares ZygOS with the Tofino implementation of `JBSQ(3)` for the 10µs exponentially-distributed service time workload using a single Xeon server. As in the previous configurations, for the R2P2 implementation each of the 16 Xeon cores, is exposed as a worker with a distinct queue to the router, listening to a different UDP port. In this experiment, the theoretical lower bound is therefore determined by $M/M/16$. We observe that `JBSQ(3)` exceeds the throughput performance of ZygOS, with no visible impact on tail latency despite the additional hop and that `JBSQ(3)` is sufficient to achieves the maximum throughput. For a service-level objective set at 150µs, R2P2 with `JBSQ(3)` outperforms ZygOS by 1.26×. The explanation is that the R2P2 server operates on a set of cores in parallel without synchronization or cache misses, whereas ZygOS has higher overheads due to protocol processing, boundary crossings, task stealing, and inter-processor interrupts.

5.5 Lucene++

Web search is a replicated, read-only workload with variability in the service time coming from the different query types, thus it is an ideal use-case for `R2P2-JBSQ`. For our experiments we used Lucene++ [56], which is a search library ported to serve queries via either HTTP or R2P2. A single I/O thread dispatches one request at a time to 16 Lucene++ worker threads, each of them searching part of the dataset. The experimental setup relies on 16 disjoint indices created from the English Wikipedia page articles dump [91], yielding an aggregated index size of 3.5MB. All indices are loaded in memory at the beginning of the execution to avoid disk accesses. The experimental workload is a subset of the Lucene nightly regression query list, with 10K queries that comprise of simple term, Boolean combinations of terms, proximity, and wildcard queries [57]. The median query service time is 750µs, with short requests taking less than 450µs and long ones over 10ms.

Figure 10 summarizes the experiment results for running Lucene++ on a 16-server cluster, each using 16 threads. The `NGINX-JSQ` and `HTTP-DIRECT` experiments rely on 1568 persistent TCP client connections. First, we observe that `HTTP-DIRECT` over TCP and `RANDOM` over R2P2 which are multi-queue models, have higher tail-latency. Then, we see that `NGINX-JSQ` and `SW-JBSQ(1)` on R2P2 deliver the same throughput; system and network protocol overheads are irrelevant for such coarse-grain workload. Also, $n = 1$ is enough to get maximum throughput, given the longer average service time. `SW-JBSQ(1)` delivers that throughput via the optimal single-queue implementation, with a significant impact on tail latency. As a result, R2P2 lowers the 99th percentile latency by 5.7× at 50% system load over `nginx`.

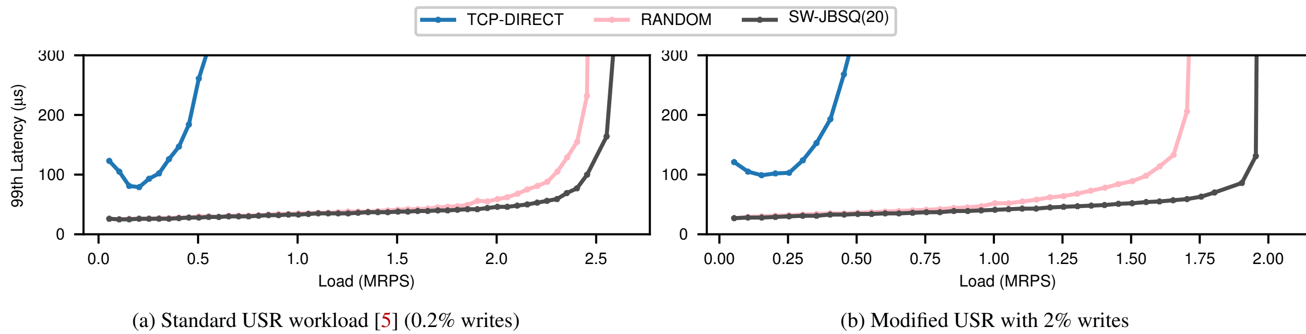
5.6 Redis

Redis [78] supports a master/slave replication scheme with read-only slaves. We ported Redis on R2P2 and ran it on DPDK for the Facebook USR workload [5]. We used the `sticky` R2P2 policy (see §3) to direct writes to the master node and we load balance reads across the master and slave nodes, based on the `RANDOM` and the `JBSQ` policy. Redis has sub-µs service times. Thus, to achieve maximum throughput we had to increase the number of tokens to 20 per worker (`SW-JBSQ(20)`), for the software router. For the vanilla Redis over TCP clients randomly select one of the servers for read requests, while they only send write requests to the master.

Figure 11a shows that R2P2, for an SLO of 200µs at the 99th percentile, achieves 5.30× better throughput for the USR workload over vanilla Redis over TCP (`TCP-DIRECT`) because of reduced protocol and system overheads, while `SW-JBSQ(20)` achieves slightly better throughput than `RANDOM` for the same SLO. Figure 11b increases the write percentage of the workload from 0.2% to 2%, which increases service time variability: R2P2 `RANDOM` has 4.09× better throughput than `TCP-DIRECT`. `SW-JBSQ(20)` further improves throughput by 18%, for a total speedup of 4.8×, as a result of better load balancing decisions.

6 Related work

RPCs can be transported by different IP-based protocols including HTTP2 [10], QUIC [48], SCTP [84], DCCP [47],



(a) Standard USR workload [5] (0.2% writes)

(b) Modified USR with 2% writes

Figure 11: 99th percentile latency vs. throughput for Redis in a 4-node master/slave configuration.

or similar research approaches [4, 28, 30, 32, 63] that identify the TCP limitations and optimize for flow-completion time. Libraries such as gRPC [31] and Thrift [86] abstract away the underlying transport stream into request-reply pairs. Approaches such as eRPC [41] aim at end-host system optimizations and are orthogonal to R2P2. Load balancers proxy RPC protocols such as HTTP in software [23, 66, 69] or in hardware [1, 16, 25, 60]. R2P2 exposes the RPC abstraction to the network to achieve better RPC scheduling, and to the application to hide the complexity of the underlying transport.

Load dispatching, direct or through load balancers, typically *pushes* requests to workers, requiring tail-mitigation techniques [17, 33]. In Join-Idle-Queue [55], workers *pull* requests whenever they are idle. R2P2 additionally supports `JBSQ(n)`, which exposes the tradeoff between maximal throughput and minimal tail latency explicitly.

Task scheduling in distributed big data systems is largely aimed at taming tail-latency and sometimes depends on split-queue designs [19, 20, 44, 71, 75, 77, 92], typically operating with millisecond-scale or larger tasks. R2P2 provides the foundation for scheduling of µs-scale tasks.

Multi-core servers are themselves distributed systems with scheduling and load balancing requirements. This is done by distributing flows using NIC mechanisms [79] in combination with operating systems [24, 73] or dataplane [9, 37, 74] support. Zygos [76] and Shinjuku [40] are an intra-server, work-conserving schedulers for short tasks that rely on task stealing and inter-processor interrupts. R2P2 eliminates the need for complex task stealing strategies by centralizing the logic in the router.

Recent work has focused on key-value stores [54,59,70,78]. MICA provide concurrent-read/exclusive-access (CREW) within a server [54] by offloading the routing decisions to the client, while hardware and software middleboxes [39, 53, 67] or SDN switches [13, 15] enhance the performance and functionality of key-value stores in-network. RackOut extended the notion of CREW to rack-scale systems [68]. R2P2 supports general-purpose RPCs not limited to key-value stores, together with a mechanisms for steering policies which can be used to implement CREW both within a single server and across the datacenter.

Finally, R2P2 adheres and encourages the in-network compute research path by increasing the network visibility to application logic and implementing in-network scheduling. Approaches leveraging in-network compute include caching [39, 53], replicated storage [38], network sequencing [51, 52], DNN training [81, 82], and database acceleration [50].

7 Conclusion

We revisit the requirements to support µs-scale RPCs across tiers of web-scale applications and propose to solve the problem in the network by making RPCs true first-class citizens of the datacenter. We design, implement and evaluate a proof-of-concept transport protocol developed specifically for µs-scale RPCs that exposes the RPC abstraction to the network and at the endpoints. We showcase the benefits of the new design by implementing efficient, tail-tolerant µs-scale RPC load-balancing based on a software router or a programmable P4 ASIC. Our approach outperforms standard load balancing proxies by an order of magnitude in throughput and latency, achieves close to the theoretical optimal behavior for 10µs tasks, reduces the tail latency of websearch by 5.7× at 50% load, and increases the scalability of Redis in a master-slave configuration by more than 4.8×.

Acknowledgements

We would like to thank Katerina Argyraki, Jim Larus, the anonymous reviewers, and our shepherd Mahesh Balakrishnan on providing valuable feedback on the paper. Also, we would like to thank Irene Zhang, Dan Ports and Jacob Nelson for their insights on R2P2. This work was funded in part by a VMWare grant and by the Microsoft Swiss Joint Research Centre. Marios Kogias is supported in part by an IBM PhD Fellowship.

References

[1] A10 Networks. https://www.a10networks.com/.

[2] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, 2012.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[6] Barefoot Networks. Tofino product brief. https://barefootnetworks.com/products/brief-tofino/, 2018.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[8] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[9] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[10] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[11] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[13] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load balancing memcached traffic using software defined networking. In *Proceedings of the 2017 IFIP Networking Conference*, pages 1–9, 2017.

[14] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[15] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. AppSwitch: Application-layer Load Balancing within a Software Switch. In *Proceedings of the 1st Asia-Pacific Workshop on Networking (APNet)*, pages 64–70, 2017.

[16] Citrix Netscaler ADC. https://www.citrix.com/products/netscaler-adc/.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[18] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[19] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 497–509, 2016.

[20] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[21] Data plane development kit. http://www.dpdk.org/.

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[23] 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 *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[24] Epollexclusive kernel patch. https://lwn.net/Articles/667087/, 2015.

[25] F5 Networks, INC. https://f5.com/.

[26] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[27] Fibre channel protocol. https://en.wikipedia.org/wiki/Fibre_Channel_Protocol.

[28] Bryan Ford. Structured streams: a new transport abstraction. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pages 361–372, 2007.

[29] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 174–178, 1999.

[30] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[31] gRPC. http://www.grpc.io/.

[32] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[33] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2017.

[34] HAProxy DSR. https://www.haproxy.com/blog/layer-4-load-balancing-direct-server-return-mode/.

[35] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 161–175, 2015.

[36] Intel Corp. Intel 82599 10 GbE Controller Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[37] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–129, 2017.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2017.

[40] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µs-scale tail latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[41] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 295–306, 2014.

[43] 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 Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, 2016.

[44] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 485–497, 2015.

[45] Marios Kogias and Edouard Bugnion. Flow Control for Latency-Critical RPCs. In *Proceedings of the 2018 SIGCOMM Workshop on Kernel Bypassing Networks*, KBNets'18, pages 15–21. ACM, 2018.

[46] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[47] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. Updated by RFCs 5595, 5596, 6335, 6773.

[48] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. 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 *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 183–196, 2017.

[49] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[50] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. The Case for Network Accelerated Query Processing. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[51] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[52] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 467–483, 2016.

[53] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44, 2016.

[54] 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 Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[55] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval.*, 68(11):1056–1071, 2011.

[56] Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[57] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench.

[58] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[59] Memcached. https://memcached.org/.

[60] 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 ACM SIGCOMM 2017 Conference*, pages 15–28, 2017.

[61] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[62] In-memory mongodb. https://docs.mongodb.com/manual/core/inmemory/.

[63] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[64] Nginx. https://www.nginx.com/.

[65] NGINX DSR: IP Transparency and Direct Server Return with NGINX and NGINX Plus as Transparent Proxy. https://www.nginx.com/blog/.

[66] NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[67] 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 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[68] Stanko Novakovic, Alexandros Daglis, Dmitrii Ustiugov, Edouard Bugnion, Babak Falsafi, and Boris Grot. Mitigating Load Imbalance in Distributed Data Serving with Rack-Scale Memory Pooling. *ACM Trans. Comput. Syst.*, 36(2):6:1–6:37, 2019.

[69] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 125–139, 2018.

[70] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[71] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[72] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed on 20.09.2018.

[73] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert Tappan Morris. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference*, pages 337–350, 2012.

[74] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[75] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 293–306, 2010.

[76] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[77] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 2016 EuroSys Conference*, pages 36:1–36:15, 2016.

[78] Redis. https://redis.io/.

[79] Microsoft corp. receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[80] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[81] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*, pages 150–156, 2017.

[82] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR*, abs/1903.06701, 2019.

[83] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[84] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335, 7053.

[85] 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 *Proceedings of the 33rd International Conference on Very Large DataBases (VLDB)*, pages 1150–1160, 2007.

[86] Apache thrift. https://thrift.apache.org/.

[87] 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)*, pages 18–32, 2013.

[88] Voltdb. https://www.voltdb.com/.

[89] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[90] Adam Wierman and Bert Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 60(5):1249–1257, 2012.

[91] The english wikipedia page article dump. https://dumps.wikimedia.org/enwiki/20180401.

[92] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[93] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologie (FAST)*, pages 167–180, 2016.

[94] 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 *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.

Lancet: A self-correcting Latency Measuring Tool

Marios Kogias[1] Stephen Mallon[2] Edouard Bugnion[1]

[1]*EPFL, Switzerland* [2]*University of Sydney*

Abstract

We present LANCET, a self-correcting tool designed to measure the open-loop tail latency of µs-scale datacenter applications with high fan-in connection patterns. LANCET is self-correcting as it relies on online statistical tests to determine situations in which tail latency cannot be accurately measured from a statistical perspective. The workload configuration, the client infrastructure, or the application itself could, under circumstances, prevent accurate measurement. Because of its design, LANCET is also extremely easy to use. In fact, the user is only responsible for (i) configuring the workload parameters, *i.e.,* the mix of requests and the size of the client connection pool, and (ii) setting the desired confidence interval for a particular tail latency percentile. All other parameters, including the length of the warmup phase, the measurement duration, and the sampling rate, are dynamically determined by the LANCET experiment coordinator.

When available, LANCET leverages NIC-based hardware timestamping to measure RPC end-to-end latency. Otherwise, it uses an asymmetric setup with a latency-agent that leverages busy-polling system calls to reduce the client bias.

Our evaluation shows that LANCET automatically identifies situations in which tail latency cannot be determined and accurately reports the latency distribution of workloads with single-digit µs service time. For the workloads studied, LANCET can successfully report, with 95% confidence, the 99th percentile tail latency within an interval of $\leq$ 10µs. In comparison with state-of-the-art tools such as Mutilate and Treadmill, LANCET reports a latency cumulative distribution that is ~20µs lower when the NIC timestamping capability is available and ~10µs lower when it is not.

1 Introduction

Today's webscale datacenter applications such as search, social networking, and e-commerce all rely extensively on the decomposition of online, data-intensive queries into smaller subqueries that process data directly from the memory of hundreds or thousands of tightly-interconnected servers to ensure service-level objectives, scalability and availability [4,11,12,33,34]. The combined advancements in hardware technology (*e.g.,* 10-100Gbps NICs, cut-through switches, NVMe), system software (*e.g.,* dataplanes [6,49]), and data management systems (*e.g.,* in-memory databases and key-value stores [17,47,48,55]) now allow µs-scale interactions between application components [5]. The increased number of components involved in a single query and the extensive use of high fan-in, high fan-out patterns have shifted the performance focus to tail-latency considerations [11].

This emerging µs-scale computing era is characterized by new key performance metrics such as the tail-latency service-level objective (SLO), *e.g.,* 99th *percentile* $\leq$ 500µs [6,29]. To put this into perspective, 500µs is one order of magnitude longer than an in-memory relational database processing TPC-C [55] and two-to-three orders of magnitude longer than basic operations on a key-value store [17,35,48]. Yet, it is shorter than an operating system quantum, a TCP retransmission timeout, or the DVFS governor's reaction time [36]. This requires complete rethinking of traditional assumptions about systems, stacks, protocols, and applications [5,17,48,53]. A large body of research focuses on the systematic characterization and reduction of tail latency effects [6,13,14,23,27–29,34,36,50].

While throughput can easily be measured, tail latency is harder to capture and characterize in a statistically meaningful manner, as it depends on a number of factors beyond the workload itself. These factors include the choice of a tool with overheads and biases, its precise configuration, and the experimental methodology. The literature describes many pitfalls specific to latency, *e.g.,* Treadmill [58] discusses situations in which: (i) the inter-arrival request distribution does not match the production environment; (ii) the measuring methodology silently masks some tail behaviors; (iii) the measuring tool affects the measured end-to-end latency because the measuring granularity is too coarse-grained or because the clients are overloaded.

This matches our own experience in building and evaluating multiple research systems for µs-scale computing [6,30,

50, 51], which used either modified versions of Mutilate [33] or home-grown latency-measurement and load-generation tools. While these tools measured the tail latency of our systems as a function of the load, we were required to unscientifically tweak a large number of workload and system parameters in an ad-hoc manner by: (i) repeatedly increasing the number of load-generating clients until stability; (ii) repeatedly increasing the number of outstanding requests (*e.g.,* number of connections) until the tail-latency diverges at saturation, as expected in an open-loop process; (iii) and last, but not least, running each experiment "longer" with the hope of reducing result jitter.

This paper introduces LANCET, a self-correcting latency measurement tool designed to measure, in a statistically sound manner, the end-to-end tail latency of remote procedure calls in a testing environment. LANCET is self-correcting as it relies on on-line statistical tests to determine situations in which tail latency cannot be accurately measured. This includes situations when (i) the workload configuration, and in particular the number of client connections, leads to closed-loop behavior; (ii) the infrastructure (*e.g.,* number of machines) cannot generate the desired load without introducing client bias; (iii) the service time of the workload itself is heavy-tailed distributed.

Because it relies on statistical methods within its control system, LANCET is also easier to use than existing tools. While the scientist specifies the infrastructure used for an experiment (*e.g.,* number of client machines), and the workload itself (*e.g.,* mix reads and writes, distribution of keys and values, number of client connections, maximum number of outstanding requests per connection *etc.*), LANCET then automatically determines, using statistical tests, what can be measured and at which confidence interval. LANCET's control system internally sets additional experimental parameters such as the duration of the experiment and its warmup phase.

Finally, LANCET relies on state-of-the-art, hardware-based measurement techniques that combine NIC timestamping in hardware and userlevel matching of packets to RPCs. This approach noticeably eliminates the client bias, and increases the accuracy of individual measurements without creating a long-term dependency on immature kernel-bypass protocols stacks and libraries.

This paper contributes the methodology, design, and implementation of LANCET, with the following novel features:

- LANCET measures the open-loop tail latency of a workload using only two user-provided parameters: the target load level and the desired confidence interval at the target tail percentile. For this, it relies on proven statistical methods such as hypothesis testing to configure the experiment methodology parameters.
- LANCET is self-correcting and reports "N/A" when no statistically-sound tail latency can be measured. This can be due to limitations in workload specification, client infrastructure, or because the service time distribution has high variability.
- LANCET clearly separates (i) the methodological considerations, implemented by the LANCET controller, (ii) the measurement tool, implemented by a combination of agents, and (iii) the workloads and application-level protocol support, implemented in an extensible manner by the LANCET agent's internal API.
- LANCET is designed with stability and production deployment in mind, with a focus on Ethernet-based protocols. It therefore uses exclusively the standard Linux kernel-based implementations of networking protocols. For applicable NICs, LANCET supports hardware-based timestamping to measure TCP-based RPC latencies for improved measurement accuracy. Our work demonstrates that kernel-bypass is not necessary to achieve precise µs-scale client-side measurements.

Our evaluation of LANCET with workloads with synthetic service times demonstrates that it (i) automatically identifies the right number of samples necessary for the target experiment accuracy and result convergence; (ii) accurately reports the latency distributions for workloads with service time as short as $\bar{S} = 1\mu s$; and (iii) provides substantially more accurate results than Treadmill [58] and Mutilate [33], state-of-the-art tail latency measurement tools.

LANCET is open-source and can be found at `https://github.com/epfl-dcsl/lancet-tool`. The rest of the paper is structured as follows. We discuss the necessary background (§2), analyze a latency experiment (§3), and discuss the design (§4) and implementation of LANCET (§5). We then evaluate LANCET (§6) and compare our methodology to existing tools (§7), and conclude (§8).

2 Background

The accurate measurement of the latency of any software application serving RPCs requires the appropriate combination of metrics, tools, workloads and experimental methodology: (i) the metrics determine which type of latency is being measured for a certain load, whether mean, median, tail (*e.g.,* 99th percentile), or the empirical cumulative distribution function (ECDF); (ii) the choice of tool determines the precision of the benchmark; (iii) the choice of workload determines the degree of realism, generality, and relevance of the experiment; (iv) the choice of methodology determines the overall soundness of the results, their accuracy, and reproducibility.

The high-level process is straightforward: the tool acts as an RPC client which generates requests for the system(s) under test. The tool timestamps requests and corresponding replies to determine the end-to-end latency. The requests themselves are determined in a workload-specific manner (*e.g.,* a mix of

get and set with a specific distribution of keys). The individual request inter-arrival time typically follows a Poisson process for a given rate λ. For a fixed rate λ, scientists typically report the full ECDF or the complementary cumulative distribution or tail distribution (CCDF), often on a log scale, to highlight the tail latency levels (99th, 999th, *etc.*). To study the impact of load on latency, scientists repeat the fixed-rate experiment for different λ and report the tail latency as a function of the load [6, 14, 22, 29, 29, 33, 37, 50, 58]. Finally, for dynamic experiments that mimic daily datacenter patterns, the tool dynamically adjusts λ according to a diurnal (or accelerated) time pattern [4, 6, 51, 56].

2.1 Taxonomy of tools

We attempt to make a taxonomy of the existing tools for generating load and measuring latency, the techniques used and the main design decision.

Packet vs. RPC generators: At the highest level, latency measuring tools can be easily classified into packet generators, which measure a network device or a network function, and application RPC generators, which measure a server.

Packet generators use stateless network packets to measure the throughput and the latency of datacenter network equipment such as switches and routers as defined in RFCs [7, 39]. These tools can be implemented to achieve different levels of precision in software. For example, MoonGen [19], TRex [8], and netperf [46], rely on hardware timestamping facilities in modern NICs (*e.g.,* MoonGen) or use custom hardware appliances such as Spirent [54] or IXIA [24].

Application RPC generators measure the latency of client-server interactions using protocols such as http or memcached's binary protocol, typically implemented on top of TCP or RDMA connections. These tools provide advanced workload-generation capabilities. For example, Mutilate [33, 45] can model Facebook's various uses of key-value stores [4], YCSB [10, 57] can generate a Zipfian distribution of keys, and CloudSuite [9, 20] offers a mix of applications. For the rest of the paper, we will focus on RPC generators.

Open-loop vs. Closed loop: There are two main ways to control the flow of requests to the target. An open-loop system models $n = \infty$ clients that send requests to the target according to a rate λ and an inter-arrival distribution, *e.g.,* Poisson. A closed-loop system bounds the maximum number of possible outstanding requests at any given time. The distinction between an open and a closed loop system is a property of a specific deployment and the same system can be deployed under different scenarios, *e.g.,* a key-value store may serve only a few blocking clients (*i.e.,* closed-loop) or thousands of application servers, which is best modelled as open-loop. Testing for the right scenario is crucial because open-loop systems can lead to large queuing, and thus longer tail latencies, whereas closed-loop tail latencies are typically bounded by the number of possible outstanding requests. Tools such as Treadmill and Mutilate are open-loop systems, while others such as YCSB are closed-loop systems.

Generating the necessary load: Precise tools are typically used to evaluate the benefit of innovations in new hardware, protocol designs, kernel bypass architectures, networking stacks, operating system configurations, or applications. Leading research systems today can deliver high-throughput solutions that easily scale to millions of requests per second, even on commodity hardware. As a consequence, the load-generation and latency-measuring tools, which typically run on reference vanilla Linux infrastructure, must be distributed on multiple client machines to saturate a single server [6, 33, 50].

Multi-machine setups follow two basic design patterns. First, in symmetric generators, all client machines generate load and measure latency. Then, an external agent accumulates and processes the collected results to report the aggregated verdict. This category includes YCSB [10], Treadmill [58], CloudSuite [20], memaslap [43], *etc.* Unfortunately the open-source versions of these tools provide no coordinator or aggregator script to run them in a distributed fashion.

Second, the asymmetric design splits the client machines between load-generating and latency-measuring. The bulk of the load is generated by client machines that generate requests according to a specific inter-arrival distribution, *e.g.,* Poisson, in an open-loop manner without measuring latency, while a separate, dedicated client machine makes closed-loop requests to the same server and measures its latency. By reducing the system load on the latency-generating thread, such tools reduce client bias in the measurement. Mutilate [33] is the most well-known tool in that category.

Point of measurement: Latency can be measured at different points in the system resulting in different levels of accuracy. This includes the actual wire, the NIC, the Ethernet driver, the in-kernel socket layer, or the application itself. The point of measurement has a large impact on precision. According to Primorac *et al.* [52], (i) the packet generators using hardware-based NIC timestamping such as MoonGen can accurately measure the latency of stateless network functions up to the 99.99^{th} percentile whereas (ii) the best software solution relying on kernel bypass can only measure up to the 99th percentile, and (iii) the solutions relying on the traditional networking stack should not be used at all for µs-scale latency measurements.

NIC-based timestamping is available on mainstream NICs. Intel NICs, such as 10Gbe 82599 and x54, or 40Gbe x710, implement hardware timestamping only to support IEEE 1588 Precision Time Protocol [18]. This restricts the type and amount of packets that can be hardware-timestamped. The MoonGen packet generator takes advantage of this precise, yet

restrictive mechanism. The Mellanox NICs, *e.g.,* ConnectX-4 [42] or newer, offer general-purpose hardware timestamping support to all incoming and outgoing packets. The Linux kernel provides support for hardware timestamping via the Linux socket interface, yet deriving RPC timestamps from packet timestamps is challenging, as later described.

Another way to increase precision and reduce jitter is to leverage kernel bypass and NIC polling at the client. Tools such as MoonGen and T-Rex use the DPDK [16] toolkit for better performance and precision. Unfortunately, kernel bypass limits application and protocol support, and requires using less-proven protocol stacks as part of the experiment.

Reporting results: From a methodology perspective, most tools depend on histograms to compute latency percentiles, thus avoiding keeping all the recorded latency samples. Histograms with fixed bucket sizes, as used by Mutilate, can affect the reported results by masking tail phenomena, if not configured properly. Some tools such as Treadmill [58] propose a user-defined calibration phase to determine the bucket allocation. Other tools, such as TailBench [29], use dynamic histograms, whose bucket sizes change over the execution of the experiment. Finally, few tools, *e.g.,* Mutilate, allow collecting all latency samples and save them in a file to be used for plotting the ECDF.

2.2 Configuration burden

Load generators put the methodological burden on the scientist who configures it. For example, a scientist using Mutilate must first determine the time for each load experiments (default=5s), which must be long enough to be statistically sound; then specify the number of machines, threads and overall number connections for the load-generating agents, and the maximum number of outstanding requests per connection; and finally specify the configuration of the latency-measurement agent, which operates as a closed-loop with one outstanding request a time.

This configuration setup has subtle implications as (i) increasing the number of machines reduces client bias [58]; (ii) increasing the number of open sockets reduces the throughput of the server because of operating system overheads [6]; (iii) increasing the maximum number of outstanding requests per socket allows for batching and increases throughput; (iv) the product of the number of connections $\times$ the number of outstanding requests must be larger than the bandwidth-delay product of the workload if the scientist wishes to measure the open-loop tail latency of the service.

Figure 1 illustrates the challenge via the study of an out-of-the-box memcached/Linux deployment with Mutilate configured with 320 and 144 connections with one outstanding request each. We report the 99th tail latency as a function of the load. The orange curve (144 Connections) operates as a closed-loop, with the clients unable to generate the target rate,

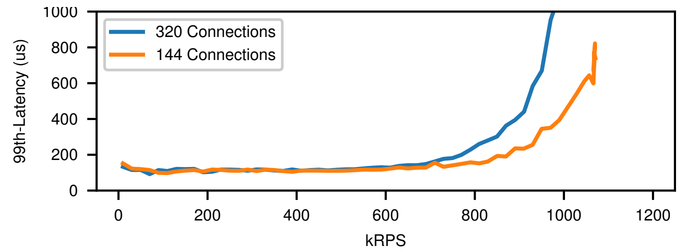


Figure 1: 99th percentile latency for memcached USR measure with Mutilate with 144 and 320 connections with 1 outstanding request per connection.

and without ever saturating the server. The lower reported tail latency is merely a reflection of the limited number of outstanding requests. This experiment can lead to false conclusions, *e.g.,* on the maximum throughput that meets a SLO (*e.g.,* $\leq 300\mu$s).

2.3 Statistics Background

This section provides the sufficient background to understand our use of statistical methods in LANCET.

Hypothesis testing: Statistical testing follows a specific thought process. Initially, the statistician formulates a null hypothesis implying that there is no relation between two populations and the observations are the results of pure chance. She then identifies a test statistic that can assess the truth of the null hypothesis and computes the p-value. p-value gives the probability of the given test statistic resulting in the observed value if the null hypothesis is true. The smaller the p-value, the stronger the evidence against the null hypothesis. Finally, she compares the p-value to the α value, which corresponds to the level of confidence. If the p-value is less than α, she rejects the hypothesis and therefore conclude that the effect she observed was not due to random chance.

LANCET uses the following tests. First, the **Anderson-Darling** test checks whether a group of samples comes from a certain probability distribution and was chosen because it is less sensitive to outliers compared to similar tests, *e.g.,* the Kolmogorov-Smirnov test [31]. We use that to validate the inter-arrival request distribution. Second, the **Augmented Dickey Fuller** test [15] checks a series of samples for stationarity. We use the ADF test to determine the duration of the warm-up phase and whether the experiment results change over time. Finally, we use the **Spearman** rank correlation coefficient and the associated p-value to check if a series of samples is autocorrelated when checking for *iid*-data.

IID-data: Most types of hypothesis testing or general statistical processing, such as the calculation of confidence intervals, require samples that are *independent and identically distributed (iid)*. When running a latency experiment, latency

Latency Experiment Concerns			
Workload		**Methodology**	**Measuring Tool**
transport protocol	connection balance	system stability	workload-compliant
application protocol	open/closed queueing	unbiased result processing	methodology-compliant
request types and ratio	outstanding requests/connection	result convergence	measuring bias free
connection count	inter-arrival distribution	distribution coverage	

Table 1: Classification of concerns related to running a latency experiment into workload, methodology, and measuring tool-specific components. We advocate that the Workload column has to be user defined, while Methodology and Measuring Tool columns have to be handled systematically by the measuring framework.

samples are naturally identically distributed since they come from the same target server. Sample independence, though, is challenging to meet because of queuing effects. The end-to-end latencies of two requests that are queued back-to-back are dependent because the latency of the latter request includes the service time of the prior. While independence cannot be taken for granted, it can be tested, with autocorrelation being the standard way to check independence for a series of samples.

Confidence Intervals: We focus here on the confidence intervals for tail latency of a *single* execution, assuming that the system environment remains identical and stable during the entire experiment. The confidence intervals for a distribution's percentiles can be computed in closed form when the data are *iid*. The formula identifies, with a certain level of confidence, two threshold values that belong to the collected samples, between which the value for the specific percentile is expected to be found. Formulas 1, 2 give the indices of those two threshold values in the sorted of collection of samples [31] for a certain confidence level γ.

$$j \approx \lfloor np - \eta\sqrt{np(1-p)} \rfloor \quad (1)$$

$$k \approx \lceil np + \eta\sqrt{np(1-p)} \rceil + 1 \quad (2)$$

where n is the number of samples, p is the percentile, and η is defined as $N_{0,1} = \frac{1+\gamma}{2}$. For example, for 10,000 *iid* samples ($n = 10000$), the confidence interval for the 99-th percentile with 95% confidence ($\gamma = 0.95$, so $\eta = 1.96$) will be between the values with indices $j = 9880$ and $k = 9921$.

Note that determining confidence across different executions of the same experiment is challenging as the system's boot-time and application initialization can have a persistent effect on performance, leading to the hysteresis problem described in Treadmill [58].

3 Experiment Decomposition

Our goal is to build a latency-measuring tool that is precise and simplifies the configuration burden discussed in §2.2, with the explicit objective to identify situations in which the configuration cannot lead to a statistically meaningful result.

Table 1 classifies concerns related to a latency experiment into three main categories: workload, methodology, and measuring tool. These concerns often correspond to user-defined parameters in most of the existing tools and can be easily misconfigured. This decomposition will guide the design of modular, self-correcting latency-measuring tools. We claim that the workload-specific parameters have to be user defined, otherwise the experiment is insufficiently described. The methodology and measuring tool concerns have to be systematically managed by the measuring framework to reduce the pitfalls induced by the user misconfiguration.

Workload: The first aspect of a latency experiment is the actual workload and a large set of the configuration parameters refer to the workload specification. The experiment workload is both application- and deployment-specific, meaning that the same application should be tested differently if the deployment environment is also different. The workload includes the application specific parameters (*e.g.,* `get:set` ratio, request size distributions, TPC-C request mix, *etc.*), the application-level protocol (*e.g.,* `HTTP`, binary memcached, *etc.*) and the network-level protocol (*e.g.,* UDP vs. TCP). The definition of the workload also includes the client assumptions, *i.e.,* the number of expected client connections, the maximum number of outstanding requests per connection, and whether clients operate in an open- or a closed-loop system.

Critically, the specification of the workload is independent of the measuring tool , but affects the results, which could lead to unrealistic or wrong conclusions. For example, one cannot meaningfully report the open-loop tail latency of a workload with an insufficient number of connections, or insufficient outstanding requests per connection.

Measuring methodology: The second aspect of a latency experiment is the methodology, which describes how the latency samples are collected and processed. Examples of configuration parameters that are relevant to the methodology are the experiment duration, the number of collected samples, and the number and size of the histogram buckets. Reducing the number of configuration parameters related to methodology is a major goal of our design.

Regarding the latency sample collection, a good method-

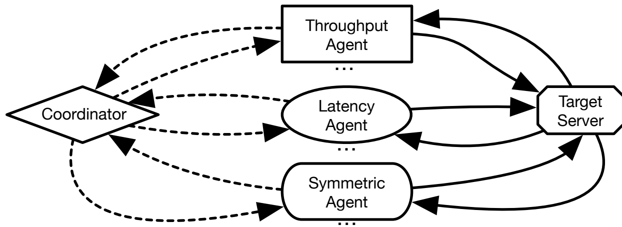


Figure 2: Lancet's architecture depicting a coordinator (C), throughput agents (TA), latency agents (LA), symmetric agents (SA), and the target server under test. The dashed arrows correspond to the LANCET API while the solid ones are application RPCs

ology should first ensure that the system under test is in a steady state to avoid measuring transient phenomena. Then it should ensure that the collected results converge and that all desired tail behaviors are covered. Finally, during the result processing, it should avoid adding statistical bias, *e.g.,* by the misconfiguration of histograms.

Measuring tool: Finally, the last part of a latency experiment is the actual client software used to collect the latency samples. Examples of parameters related to the tool are the number of client machines or threads used in the experiment, and whether hardware or software timestamping is used. The tool should be able to implement the specific methodology, generate the target workload accurately, and measure latency without adding too much client bias.

4 Design

4.1 LANCET infrastructure

Figure 2 shows the basic LANCET overview, which splits the methodology from the actual measuring tool and workload generator according to §3. LANCET is a by-design distributed tool that consists of a coordinator (C) and various measuring agents. The coordinator is in charge of the experiment methodology (see §4.3) and communicates with the agents over the LANCET API (Table 2). The measuring agents drive the workload via application RPCs generated based on application-specific random distributions. The agents also measure latency precisely, identify cases of workload violations, and run statistical tests.

Figure 3 describes a typical agent state transitions triggered by the coordinator via the API for a fixed-load experiment. From an idle state (*Idle*), the agent transitions into the loading phase (*Load*), where it attempts to issue *l* requests per second to the server. During that period the agent does not record latency. The agent eventually transitions into the measurement phase (*Measure*) specified by a sampling rate (*sr*) and a number of latency samples to collect (*s*). The agent can

Request Type	Request Params	Reply
start_load	load (rps)	ACK
start_measure	#samples sampling rate(sr)	ACK
get_throughput	None	Throughput (rps) Correct IA (T/F)
get_latency	None	Latency CI Stationary (T/F) IID (T/F, sr)
exit	None	ACK

Table 2: The LANCET coordinator API with the information returned by the agents on each call. For the `get_throughput` and `get_latency` requests, the agents also reply information related to the Inter-Arrival distribution (IA), the latency Confidence Intervals (CI), and whether the collected samples are stationary and *iid*. If they are not *iid*, the reply contains the target sampling rate necessary to get *iid* data.

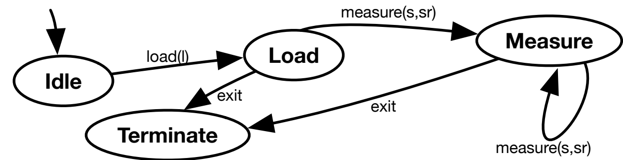


Figure 3: Lancet agent's state transition. Arrows represent messages from the coordinator.

stay in that state while the *sr* and *s* parameters can change. Finally, the coordinator decides to terminate the experiment (*Terminate*) via an `exit` message. At any point in time, while the agent is in the *Load* or *Measure* phase the coordinator can ask for the current throughput and latency.

4.2 Measurement options

Figure 2 shows that LANCET implements three agent types, selected to match the capabilities of the available hardware, the measuring methodology and the target experiment granularity. This way LANCET can support both symmetrical and asymmetrical deployments described in § 2.1.

LANCET uses the asymmetrical model when the latencies are captured in software. This model reduces jitter by dedicating cores and even machines to only measure latency. The drawback is that the experiment collects fewer samples per time period. Furthermore, special care must be taken to ensure that the collected samples are representatives of the workload. For example, a latency agent should open multiple connections (*i.e.,* emulate multiple clients) to ensure that a server configured with an RSS NIC will use all cores.

LANCET uses the symmetrical model when the NIC offers the capability to timestamp all incoming and outgoing Ethernet frames and the Linux operating system exposes the infor-

mation to userspace (v4.14+ kernels). LANCET associates the hardware timestamping of packets to the end-to-end latency of RPCs. This is not straightforward because of the inherent mismatch between the stream-oriented TCP protocol and the message-oriented RPCs. Implementation details follow in §5.

4.3 LANCET's self-correcting methodology

LANCET's primary contribution is its novel, self-correcting methodology which follows the experiment decomposition and split of concerns described in Table 1, and tries to systematically and based on statistics, identify: (i) when the server is in a stable state to start measuring latency (managed by the user-defined warm-up time in other tools); (ii) if the collected latency samples converge and whether tail phenomena are fully covered (controlled by the user-defined experiment duration in other tools); (iii) how to process the collected samples and report latency without introducing statistical bias (histograms are mainly used for that purpose in other tools); (iv) the confidence intervals of the latency results (unlike most tools which simply report latency percentiles).

Figure 4 illustrates the state machine transitions of the coordinator when measuring the open-loop tail-latency of a server under a certain load. To run such an experiment, the scientist needs to provide, apart from the necessary workload specification (first column in Table 1), the following:

- the target load (`l`).
- the target confidence interval for a specific latency percentile, *e.g.,* 10µs interval for the 99th percentile with 95% confidence.

The output of such an experiment will be either the tail-latency percentiles with the corresponding confidence intervals or an indication that the specific experiment cannot be executed because some of the assumptions are violated, *e.g.,* the target load cannot be reached, the service time has high variability and the computed latency confidence interval is wider than the target, the client does not respect the workload specifications, *etc.*

System Stability: Initially, the methodology ensures that the target load can actually be reached before starting measuring latency, thus eliminating transient phenomena. Then, the methodology ensures that agents load the server while respecting the workload's specified inter-arrival distribution. This second confirmation is essential to avoid reporting misleading latencies. For this, every agent records the inter-transmission intervals of requests by recording request transmissions, ideally in hardware, but if necessary at the socket interface. Every agent runs an Anderson-Darling test to check whether the inter-transmission intervals follow the target inter-arrival distribution, *e.g.,* exponential in the case of Poisson inter-arrival. The controller exits the system stability step only when the load is reached according to the correct inter-arrival distribution.

Unbiased Result Processing: Each agent collects, according to the parameters (s, sr) set by the controller, s samples, each randomly sampled among the RPCs at rate of sr, *e.g.,* collecting 10,000 samples with a 1:20 sampling rate would require $\sim$200K RPCs. Sampling is necessary because the collected samples need to be *iid* to compute the confidence interval correctly. Computing confidence intervals on non-*iid* data will underestimate their size.

The *iid*-ness is confirmed or rejected by computing the autocorrelation of the collected latency samples sorted by their transmission time. To do so, latency-measuring agents compute the Spearman correlation of the collected latency samples shifted over time. We leverage the associated p-value to determine whether the correlation is significant or not. This correlation being significant implies that data that are close in time depend on each other, which is the result of them being queued back to back in the servers queue.

A way to reduce the autocorrelation is to decrease the sampling rate. The LANCET built-in parameters initialize the measuring phase with 10,000 collected samples with a first sampling rate of 1:5. If the autocorrelation is non-significant, the latency measuring agents report that the samples are *iid*. Otherwise, they report how much to reduce the sampling rate to achieve non-significant correlation. The latency measuring agents report the Pearson correlation co-efficient back to the coordinator as part of their latency results. To do so, the agents compute the autocorrelation for different lags and report the one that leads to a non-significant correlation. Based on the agents' replies, the coordinator decides whether to proceed to the next state or reduce the sampling rate accordingly if it fails to confirm *iid*-ness.

Result Stationarity: The methodology needs to identify whether the number of samples collected is sufficient for the results to converge to a stable distribution of latencies that does not change over time. To ensure stationarity, the methodology leverages an Augmented Dickey Fuller test [15]. Each latency-measuring agent sorts the collected latency results based on their transmission timestamp and runs the test. Again, the latency measuring agents report the result of the test to the coordinator. In cases where lack of stationarity is detected, the coordinator decides to increase the number of samples by 10,000 and retry. Otherwise it proceeds with the next check.

Determine the confidence interval: Finally, the methodology has to check if the results converge within the target confidence interval size. For that, we use the Formulas 1 and 2. Each latency measuring agent reports the confidence intervals for the latency percentiles to the coordinator. The coordinator ensures that the intervals from different agents

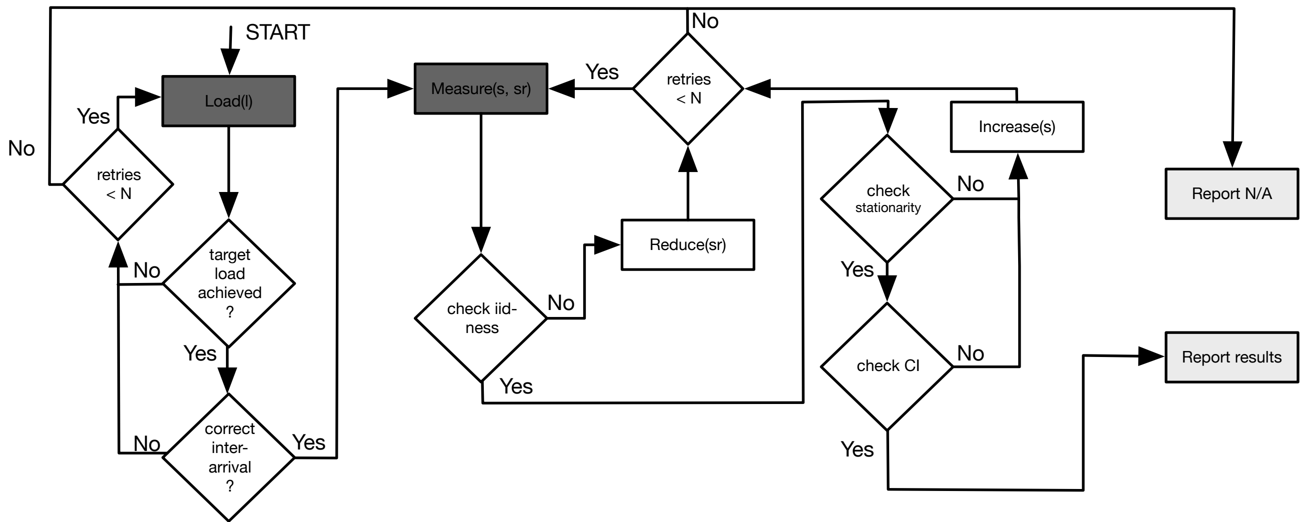


Figure 4: LANCET's experiment methodology implemented by the coordinator. Dark grey boxes correspond to messages from the coordinator to the agents. Light grey boxes show the experiment end.

are overlapping and computes their average. If the final confidence interval is wider than the user-selected target, the coordinator increases the number of samples by 10,000 and continues the experiment.

Termination: If the target confidence is reached, the coordinator finishes the experiment and reports the final latency percentiles with the equivalent confidence intervals. If the coordinator cannot reach the target confidence after a fixed number of failed retries, or if the experiment duration is above a certain threshold, it terminates the experiment, and reports that the specific experiment is not conclusive, the reasons why, along with the collected results so far.

5 Implementation

In addition to the design goals of §4, LANCET was implemented with robustness and long-term relevance in mind. LANCET is therefore built purely on functionality provided by the Linux kernel, using built-in drivers and protocol stacks. During development we identified some inconsistencies regarding hardware timestamping in the Linux kernel; our patch was merged in Linux kernel 4.19.4 [38].

The LANCET coordinator (Figure 2) is in charge of deploying the agents, communicating with them over sockets, driving their state machine according to Figure 3, and implementing the methodology of Figure 4. The coordinator is implemented in Golang. It relies on *goroutines* for easy distributed coordination and failure management, and consists of ∼1000 lines of code. From those lines, ∼300 of them implement the methodology described in § 4.3 and the rest implemented the LANCET API to communicate with the agents, manage collected results, *etc.* Thus, implementing a new coordinator logic for different experiment methodologies is relatively easy.

We implemented three different agents that can be used according to the available hardware, the measuring methodology, and the necessary experiment granularity. Our agents can achieve better measuring granularity compared to previous tools and can be used in both a symmetrical and asymmetrical deployment, independently of the available hardware. The agents are implemented in a combination of C and Python, and can be easily extended with new transport and application protocols.

Figure 5 depicts the structure of a multi-threaded LANCET agent. Each agent is split between a Python control plane and a C data plane communicating over shared memory. The Python control plane is in charge of communicating with the coordinator and performing the statistical computations. The choice of Python allowed us to take advantage of the rich Python ecosystem using libraries such as NumPy and SciPy. The choice of C for dataplane gave us direct access to low level socket APIs and reduced the client overhead. LANCET lancet currently supports TCP, UDP, and R2P2 [30] as transports, and Memcached, Redis, and HTTP as application protocols.

Throughput Agent: This agent leverages `epoll_wait` to manage connections and is in charge of loading the server without measuring latencies. It is used only in asymmetrical deployments in cooperation with one of the two following agents that can measure RPC latency.

Latency SW-timestamping Agent: This agent depends on software timestamping and does not have any hardware dependencies. It improves the measuring precision over

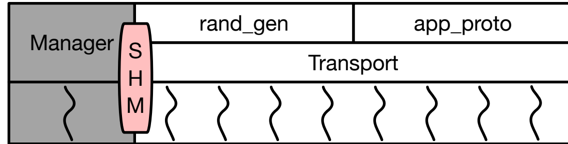


Figure 5: The structure of a LANCET agent. The grey part corresponds to the Python control plane, while the white part corresponds to the C dataplane communicating over shared memory (SHM).

other software-based tools, though, by leveraging the busy polling functionality introduced in Linux 3.11. Specifically, the `SO_BUSY_POLL` socket option allows blocking system calls to poll the NIC instead of depending on interrupts. While still dependent on userspace timestamping, this agent reduces client bias and measures latency with similar accuracy to kernel-bypass approaches. The blocking nature of this agent limits the load and the inter-arrival distribution of requests the agent can achieve. Consequently, this agent can only be used in asymmetric setups in conjunction with throughput agents that generate the necessary target system load according to the expected inter-arrival distribution.

Symmetric HW-timestamping Agent: Finally, we implemented a symmetric agent that leverages hardware timestamping to measure RPC end-to-end latency. This agent depends on the Linux kernel functionality for hardware timestamping added in kernel 4.14 for TCP. It also requires a NIC that timestamps all the incoming and outgoing packets, *e.g.*, the Mellanox ConnectX-4 [42]. The preferred deployment is based on symmetric HW-timestamping agents, as they improve on the latency agent in terms of precision and they can scale throughput while the coordinator can symmetrically collect results from all client machines, thus increasing the experiment accuracy.

The most challenging part of the implementation was the attribution of RPC latencies when requests and replies are layered on top of the stream-based TCP protocol, as used in the popular protocols, such as Memcached.

For TX timestamps, the Linux kernel provides an asynchronous API to collect timestamps, returning asynchronously one timestamp for each `sendmsg` system call. The notification is propagated to the userspace through an `EPOLERR` for the equivalent socket that is handled by `epoll_wait`. Along with the timestamp, the kernel also returns the number of the last transmitted byte this timestamp corresponds to. For example, if the first request has a size of 20 bytes, the notification will mention that this timestamp is associated with byte 20. For the second request of the same size, the notification will mention byte 40, *etc.* The same information is maintained by LANCET in userspace for validation purposes, and to deal with cases of coalescing or resubmissions.

The kernel provides a synchronous API to retrieve the RX timestamp: the RX timestamp is part of the metadata to the `recvmsg` system call, and corresponds to the receive timestamp of the frame that carried the last byte returned by the system call. The LANCET application-parsing logic leverages this information to associate timestamps to replies of variable sizes: if the content returned by `recvmsg` consists of an incomplete reply, that timestamp is ignored; if the content contains the replies to multiple requests (which is possible because of TCP's streaming nature and coalescing in the socket layer), LANCET only considers the timestamp for the last reply returned in that call. The Linux kernel coalesces `sk_buffs` internally and keeps a single timestamp per `sk_buff` corresponding to the last arrival. Consequently, earlier received responses might appear to have later receive timestamps.

Our contribution to the kernel 4.19.4 [38] guarantees that each `recvmsg` system call will return the hardware timestamp that corresponds to the last byte read. Previously, this was only the case for software in-kernel timestamping.

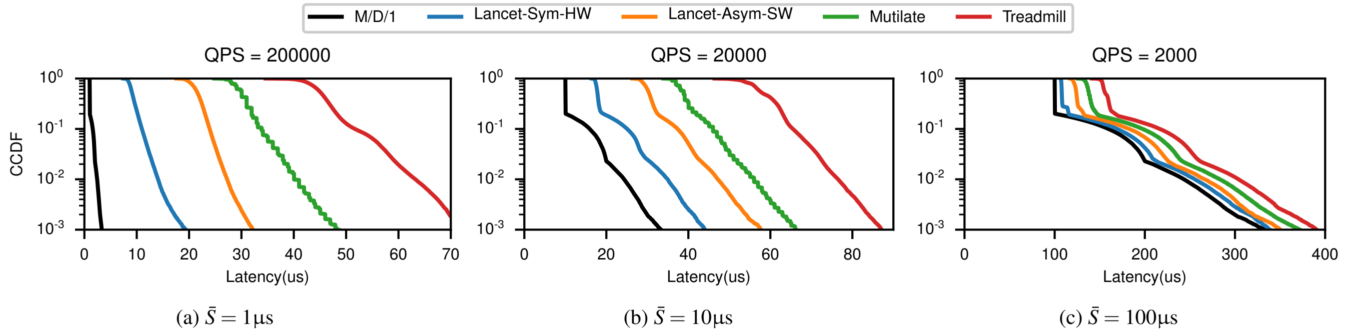
6 Evaluation

Our evaluation aims to answer three fundamental questions: (i) how does LANCET compare with existing RPC load-generating tools such as Mutilate and Treadmill (ii) how does LANCET's self-correcting methodology work in practice (iii) how LANCET performs in characterizing a server's behavior across different loads.

We answer these questions using a methodology in which the server's execution time is explicitly controlled. Doing so enables comparing the client-side measurements made by the tools to an idealized queueing theoretic model. We leverage an RPC server with synthetic service times following well-known distributions. Specifically, we tried a fixed, an exponential, and a bimodal distribution where 10% of the requests take $\sim 10\times$ longer to execute. To further reduce server-side overheads, our server uses the open-source IX operating system [6] configured with 1 CPU and adaptive batching disabled. The operating system overhead is $\sim 1\,\mu s$ of CPU execution time per request, which includes driver and network processing overheads. As baselines we use the opensource versions of Mutilate [1] and Treadmill [3]. For Treadmill, we had to make changes in order to build it for our setup.

To be able to compare with other tools, our synthetic server uses the `ascii-memcached` protocol. Clients submit `get` requests with for a 19-byte key (similarly to Facebook's USR [4]), the server spins for a configurable amount of time, and replies that that key was not found. We chose `ascii-memcached` because it is the only protocol supported by both Treadmill and Mutilate.

The idealized models correspond to the expected latency distribution, as determined by a discrete event simulation, assuming zero operating system overheads, zero network propagation delays, and zero client-side measurement overheads.



(a) $\bar{S} = 1\mu s$ (b) $\bar{S} = 10\mu s$ (c) $\bar{S} = 100\mu s$

Figure 6: Latency ECDF for an M/D/1 model and three deterministic workloads at 20% load.

For all of our experiments, we configure each client machine with 15 threads and 4 connections per thread with 1 outstanding request per connection. Also, we consider a Poisson inter-arrival distribution of requests.

6.1 Experimental setup

Our experimental setup uses 5 clients and one server machine connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are equipped with a Xeon E5-2637 @ 3.5 GHz and a Mellanox Connect-X4 NIC. The machines run an Ubuntu LTS 16.04 distribution running Linux kernel version 4.19.4. The systems are tuned to reduce jitter: all power management features, including CPU frequency governors and TurboBoost, and support for transparent huge pages, are disabled. The server is a Xeon E5-2665 @ 2.4 GHz with an Intel x520 NIC running the IX operating system.

6.2 Benefits of hardware timestamping

First, we compare the measuring granularity of LANCET with the measuring granularity achieved by Mutilate and Treadmill. For LANCET we consider both the hardware timestamping, symmetrical setup and the asymmetrical one based on the busy-polling agent. LANCET and Mutilate provide a way to run an experiment based on multiple machines, but for Treadmill there is no opensource coordinator script. Thus, we run one Treadmill instance on each client that contributes 1/5 of the load. Also, we modified Treadmill to save the collected latencies at the end of the experiment.

From a methodology perspective, we plot the latency CCDF for a deterministic service time distribution with different average service times. The load is set at 20% of the theoretical saturation, we range the average service time from $\bar{S} = 1\mu s$ to $\bar{S} = 100\mu s$. We collected 1M samples for each tool.

Figure 6 summarizes the experiment results. We observe that LANCET, for both configurations and in all three experiments, achieves lower measuring granularities when compared to the other tools because it reduces the client measuring overheads. Specifically, for $\bar{S} = 1\mu s$ hardware timestamping measures a 99th percentile tail of 14.1µs and the LANCET polling agent one of 27.3µs. Mutilate measures 40µs and Treadmill reports 63µs. Figure 6a also shows that Mutilate's line is not smooth because of the µs reported granularity, as opposed to nanoseconds reported by the other tools. Also, we see that LANCET aligns better with the theoretic results. For example, with $\bar{S} = 10\mu s$, the blue line nicely tracks the model; the offset between the two (∼10µs) is essentially due to the operating system overhead and the propagation delay. Finally, Figure 6c shows that the tools make a difference even for coarser grain tasks ($\bar{S} = 100$), where the operating system and propagation delay overheads are comparatively small.

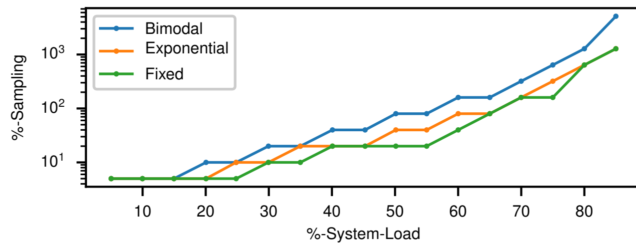
For the rest of our evaluation we will focus on the symmetric hardware-timestamping agent as it reports the most accurate results.

6.3 LANCET self-controlling dynamics

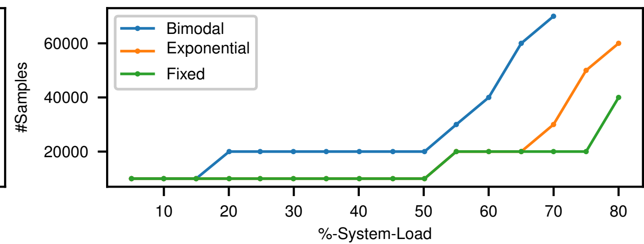
In the next series of benchmarks, we want to identify the impact of the self-correcting methodology and how the coordinator controls the experiment parameters based on the different service time distributions and the system load. To do so, we run the three different service time distributions across a variety of loads and we collect the necessary level of sampling to achieve *iid*-ness, and the number of samples necessary for a target confidence interval size of 10µs.

Figure 7a shows the sampling rate that is necessary to the unbiased processing of the results caused by queuing effects. We observe that high-dispersion workloads (*e.g.,* bimodal) and higher load levels require lower sampling rates. This is expected as increasing either service time dispersion or load level leads to more queuing, thus more dependent samples.

In Figure 7b, we set the size of the target confidence interval for the 99th percentile latency to be 10µs with 95% confidence. The figure shows the number of collected samples, as decided by the coordinator, that are required to satisfy the result target. We observe that more samples are necessary to fulfill the constraint as the load increases, since higher



(a) Required level of sampling to guarantee *iid*-ness



(b) Required number of samples to achieve the target CI of 10µs

Figure 7: Dynamics of LANCET's self-correcting methodology based on load for three service time distributions with $\bar{S} = 10\mu s$

system load leads to higher latency variability.

The bimodal distribution shows an interesting behavior of the tool: With load $> 70\%$, execution stops after the maximum number iterations ($N = 10$) but the target CI expectations can not be met. Our experiment logs showed that the collected 99th percentile latency at 75% load is 411.333µs [-5.87µs, 7.56µs] at 95% confidence; this interval is $> 10\mu s$.

We also tested LANCET's self-correcting behavior with the lognormal distribution, which is a heavy tailed distribution. LANCET terminates without ever being able to confirm results convergence(CI < 10µs for the 99-th percentile latency), even at a low load of 20%. Thus, LANCET is effective in detecting heavy-tailed service time distributions.

6.4 Inter-Arrival distribution Impact

In the following experiment we try to showcase the impact of the inter-arrival distribution on the latency results and how LANCET identifies cases of inter-arrival distribution violations. We use the fixed synthetic time distribution with $\bar{S} = 10\mu s$ and we run a latency experiment across a variety of loads with different number of connections. We disable LANCET's checks for inter-arrival distribution and we only report whether there is a workload violation. To eliminate any system interference we configure LANCET with one connection per thread, and add connections by adding client machines.

Figure 8 shows the 99-th percentile latency as a function of throughput for the different connection count configurations. The vertical lines correspond to the load level that the equivalent configuration started violating the inter-arrival distribution. We observe that once LANCET reports a violation the curves start deviating. This experiment shows that cases as the one described in Figure 1 can be avoided by LANCET's self-correcting methodology.

6.5 Server characterization

Figure 9 shows the 99th-percentile tail latency as a function of the load for three workloads. We compare LANCET with Mutilate as well as the idealized, zero-overhead theoretical model. Both tools use 5 machines – necessary to achieve the high loads required. For LANCET, we additionally report the confidence interval of each measurement. This experiment does not include Treadmill as Treadmill's open-source distribution does not support multi-machine deployments. Note that because of system overheads, the IX server cannot get close to the expected maximum load for Figure 9a which would be 1M RPS, thus we do not plot the theoretic curve.

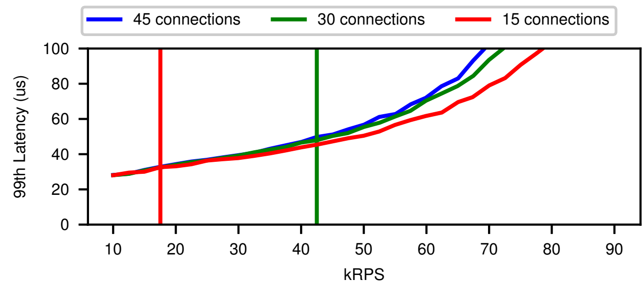


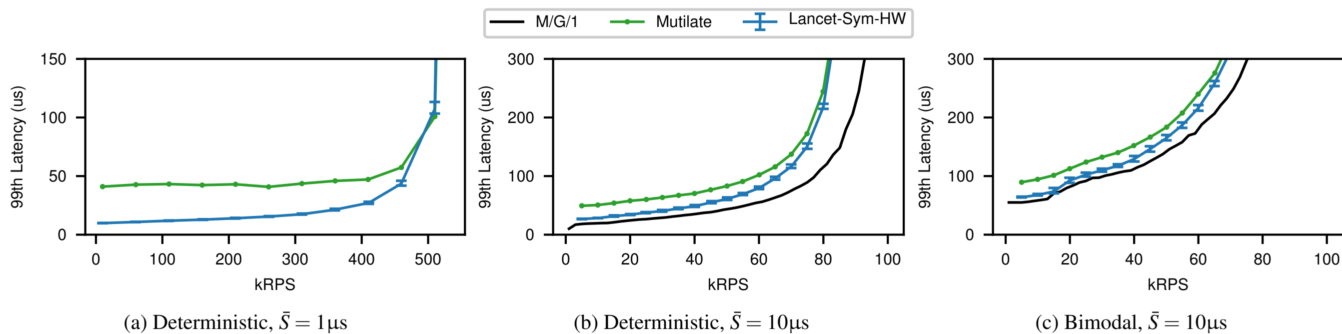
Figure 8: Impact of the inter-arrival distribution to tail-latency for a fixed with $\bar{S} = 10\mu s$. Vertical lines correspond to the load levels where Lancet reports inter-arrival distribution violations

We observe that LANCET reports latencies that closely match the idealized model across the entire load spectrum, to the point that it accurately reflects the two inflection points of the binomial distribution. We also observe how the size of the confidence intervals change across different distributions and system loads. For low loads and low service time dispersion, the interval is shorter than the maximum configured (10µs). For the bimodal distribution, the reported confidence interval is at its maximum configuration even for low loads.

7 Related Work

LANCET is one of the many contributions towards enabling reproducibility and accurate experimentation in systems research [25, 40].

µs-scale computing: Recent research focuses on µs-scale



(a) Deterministic, $\bar{S} = 1\mu s$ (b) Deterministic, $\bar{S} = 10\mu s$ (c) Bimodal, $\bar{S} = 10\mu s$

Figure 9: Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 1$ and $\bar{S} = 10$

computing [5] both in operating systems and networking and either aim to optimize [6,26,35,49,50], or attribute the sources of tail-latency [29, 33, 34, 52, 58]. LANCET does not attempt to attribute the sources of the jitter. Instead, it provides a tool to measure µs tail latency precisely on an end-to-end basis to be used in similar research efforts.

Precise measurements: RPC generators [10, 20, 33, 43, 58] use software timestamping. However, researchers need more accurate tools to evaluate system's latency, *e.g.,* ZygOS used a modified version of Mutilate based on DPDK [16], and MICA [35] used a custom version of YCSB on DPDK. Software-based packet generators [8, 19, 46] also used DPDK for increased precision [52]. Hardware-based packet generators [24, 54] provide sub µs-scale precision with little jitter [52]. Some tools repurposed the IEEE PTP feature of standard NICs to measure packet latencies [19,32,44]. LANCET is the first tool that leverages hardware-based NIC timestamping for capturing latency for RPCs over TCP with even higher precision. In addition, LANCET uses the standard Linux networking stack for all experiments, proving a more realistic simulation environment. While we used Mellanox ConnectX-4 NICs in our experiments, hardware timestamping of all packets is also available on Solarflare NICs [2].

Methodology: Although tail-latency is a widely used system metric, there is no widely accepted experiment methodology for measuring it, and usually tools are bounded to specific methodologies. LANCET attempts to split the methodology from the actual tool and reason about them separately. Measurement bias from non-determinism can be avoided via setup randomization [21, 41, 58]. Repeated runs eliminate hysteresis effects in systems [58]. Distributed benchmarking tools seek to minimize client side queuing bias by reducing the client load, in asymmetric *e.g.,* Mutilate [33], or symmetric setups [58]. LANCET's use of hardware timestamping eliminates client bias in the point of measurement. Treadmill [58] avoids issues of imbalance by leveraging a symmetric measurement model, and bias from outliers by computing interested metrics on individual instances and combining them using aggregation functions. LANCET also supports the symmetric setup to detect imbalance across client machines. Most tools use histograms to capture latencies. Treadmill determines bucket ranges during a calibration phase. YCBS [10] and Tailbench [29] have dynamic range histograms. LANCET relies on on-line sampling but keeps all sampled results to determine both the CCDF and the confidence intervals. Confidence intervals can also be used to determine statistical convergence of results [21,41]. LANCET's self-correcting controller relies on statistical tests to ensure stability and results convergence similarly to [40].

8 Conclusion

LANCET is a new latency-measuring tool designed with the explicit goal to accurately measure µs-scale tail-latencies while reducing methodological pitfalls in a principled manner. Its self-correcting methodology uses proven statistical methods to detect situations where application tail latency cannot be reliably measured. LANCET's agents uniquely leverage NIC-based timestamping to measure the end-to-end latency of TCP-based applications, completely eliminating client bias. LANCET measures latency distributions with more accuracy than popular tools such as Mutilate and Treadmill. Our evaluation with µs-scale workloads shows that it robustly self-corrects as a function of the load for workloads with challenging service time distributions.

Acknowledgements

We would like to thank our shepherd Charlie Curtsinger, Vincent Gramoli, Guillaume Jourjon, and the anonymous reviewers for their valuable comments on the paper, Prodomos Kolyvakis for his insights on the experiment methodology, Mikael Gonzalez Morales for implementing hardware timestamping for R2P2, and Christos Kozyrakis for the early discussions on the topic. This work was funded in part by a VMWare grant and by the Microsoft Swiss Joint Research Centre. Marios Kogias is supported in part by an IBM PhD Fellowship.

References

[1] Mutilate codebase (commit d65c6ef). https://github.com/leverich/mutilate.

[2] SolarFlare networking interfaces. https://www.solarflare.com/.

[3] Treadmill codebase (commit 1bf2082). https://github.com/facebook/treadmill.

[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 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[6] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[7] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544 (Informational), March 1999. Updated by RFCs 6201, 6815.

[8] Cisco Systems. T-Rex: Cisco's realistic traffic generator. https://trex-tgn.cisco.com.

[9] CloudSuite Benchmarking Suite. http://cloudsuite.ch/.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[12] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[13] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Data centers with Paragon. *IEEE Micro*, 34(3):17–30, 2014.

[14] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 127–144, 2014.

[15] David Dickey and Wayne A Fuller. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica*, 49(4):1057–72, 1981.

[16] Data plane development kit. http://www.dpdk.org/.

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

[18] John Eidson and Kang Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE.*

[19] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 15th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 275–287, 2015.

[20] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, pages 37–48, 2012.

[21] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–76, 2007.

[22] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2015.

[23] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas F. Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[24] Ixia. Ixia traffic generator. https://www.ixiacom.com.

[25] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay F. Lofstead, Kathryn Mohror, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Proceedigns of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1561–1570, 2017.

[26] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µs-scale tail latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[27] Svilen Kanev, Kim M. Hazelwood, Gu-Yeon Wei, and David M. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40, 2014.

[28] Harshad Kasture and Daniel Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 729–742, 2014.

[29] Harshad Kasture and Daniel Sánchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, 2016.

[30] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[31] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[32] Changhyun Lee, Chunjong Park, Keon Jang, Sue B. Moon, and Dongsu Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 403–415, 2015.

[33] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[34] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 9:1–9:14, 2014.

[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 Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[36] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[37] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[38] Stephen Mallon. tcp: Fix sof_timestamping_rx_hardware to use the latest timestamp during tcp coalescing. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cadf9df27e7cf40e390e060a1c71bb86ecde798b, 2018.

[39] R. Mandeville and J. Perser. Benchmarking Methodology for LAN Switching Devices. RFC 2889 (Informational), August 2000.

[40] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, Robert Ricci, and Ana Klimovic. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 409–425, 2018.

[41] David Meisner, Junjie Wu, and Thomas F. Wenisch. BigHouse: A simulation infrastructure for data center systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–45, 2012.

[42] Mellanox Corporation. ConnectX-4 NIC.

[43] Memaslap Load Generator. https://libmemcached.org/libMemcached.html.

[44] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[45] Mutilate Load Generator. https://github.com/leverich/mutilate.

[46] Netperf. http://www.netperf.org/netperf/.

[47] 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 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[48] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[49] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[50] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[51] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

[52] Mia Primorac, Edouard Bugnion, and Katerina J. Argyraki. How to Measure the Killer Microsecond. In *Proceedings of the 2017 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, pages 37–42, 2017.

[53] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[54] Spirent Communications. Spirent test modules and chassis. https://www.spirent.com/Products/TestCenter/Platforms/Modules.

[55] 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)*, pages 18–32, 2013.

[56] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[57] YCSB Load Generator. https://github.com/brianfrankcooper/YCSB.

[58] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.

Pangolin: A Fault-Tolerant Persistent Memory Programming Library

Lu Zhang
University of California, San Diego
luzh@eng.ucsd.edu

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

Abstract

Non-volatile main memory (NVMM) allows programmers to build complex, persistent, pointer-based data structures that can offer substantial performance gains over conventional approaches to managing persistent state. This programming model removes the file system from the critical path which improves performance, but it also places these data structures out of reach of file system-based fault tolerance mechanisms (e.g., block-based checksums or erasure coding). Without fault-tolerance, using NVMM to hold critical data will be much less attractive.

This paper presents Pangolin, a fault-tolerant persistent object library designed for NVMM. Pangolin uses a combination of checksums, parity, and micro-buffering to protect an application's objects from both media errors and corruption due to software bugs. It provides these protections for objects of any size and supports automatic, online detection of data corruption and recovery. The required storage overhead is small (1% for gigabyte-sized pools of NVMM). Pangolin provides stronger protection, requires orders of magnitude less storage overhead, and achieves comparable performance relative to the current state-of-the-art fault-tolerant persistent object library.

1 Introduction

Emerging non-volatile memory (NVM) technologies (e.g., battery-backed NVDIMMs [31] and 3D XPoint [30]) provide persistence with performance comparable to DRAM. Non-volatile main memory (NVMM), is byte-addressable, cache-coherent NVM that resides on the system's main memory bus. The combination of NVMM and DRAM enables hybrid memory systems that offer the promise of dramatic increases in storage performance and a more flexible programming model.

A key feature of NVMM is support for direct access, or DAX, that lets applications perform loads and stores directly to a file that resides in NVMM. DAX offers the lowest-possible storage access latency and enables programmers to craft complex, customized data structures for specific applications. To support this model, researchers and industry have proposed various persistent object systems [3, 5, 12, 20, 25, 28, 39, 47].

Building persistent data structures presents a host of challenges, particularly in the area of crash consistency and fault tolerance. Systems that use NVMM must preserve crash-consistency in the presence of volatile caches, out-of-order execution, software bugs, and system failures. To address these challenges, many groups have proposed crash-consistency solutions based on hardware [33, 34, 37, 40], file systems [4, 10, 48, 49], user-space data structures and libraries [3, 5, 12, 39, 43, 47, 50], and languages [9, 38].

Fault tolerance has received less attention but is equally important: To be viable as an enterprise-ready storage medium, persistent data structures must include protection from data corruption. Intel processors report uncorrectable memory media errors via a machine-check exception and the kernel forwards it to user-space as a `SIGBUS` signal. To our knowledge, Xu *et al.* [49] were the first to design an NVMM file system that detects and attempts to recover from these errors. Among programming libraries, only `libpmemobj` provides any support for fault tolerance, but it incurs 100% space overhead, only protects against media errors (not software "scribbles"), and cannot recover corrupted data without taking the object store offline.

Xu *et al.* also highlighted a fundamental conflict between DAX-`mmap()` and file system-based fault tolerance: By design, DAX-`mmap()` leaves the file system unaware of updates made to the file, making it impossible for the file system to update the redundancy data for the file. Their solution is to disable file data protection while the file is mapped and restore it afterward. This provides well-defined protection guarantees but leaves file data unprotected when it is in use.

Moving fault-tolerance to user-space NVMM libraries solves this problem, but presents challenges since it requires integrating fault tolerance into persistent object libraries that manage potentially millions of small, heterogeneous objects.

To satisfy the competing requirements placed on NVMM-based, DAX-mapped object store, a fault-tolerant persistent object library should provide at least the following characteristics:

1. **Crash-consistency**. The library should provide the means to ensure consistency in the face of both system failures and data corruption.
2. **Protection against media and software errors**. Both types of errors are real threats to data stored to NVMM, so the library should provide protection against both.
3. **Low storage overhead**. NVMM is expensive, so minimizing storage overhead of fault tolerance is important.
4. **Online recovery**. For good availability, detection and recovery must proceed without taking the persistent object store offline.
5. **High performance**. Speed is a key benefit of NVMM. If fault-tolerance incurs a large performance penalty, NVMM will be much less attractive.
6. **Support for diverse objects**. A persistent object system must support objects of size ranging from a few cache lines to many megabytes.

This paper describes *Pangolin*, the first persistent object library to satisfy all these criteria. Pangolin uses a combination of parity, replication, and object-level checksums to provide space-efficient, high-performance fault tolerance for complex NVMM data structures. Pangolin also introduces a new technique for accessing NVMM called *micro-buffering* that simplifies transactions and protects NVMM data structures from programming errors.

We evaluate Pangolin using a suite of benchmarks and compare it to `libpmemobj`, a persistent object library that offers a simple replication mode for fault tolerance. Compared to `libpmemobj`, performance is similar, and Pangolin provides stronger protection, online recovery, and greatly reduced storage overhead (1% instead of 100%).

The rest of the paper is organized as follows: Section 2 provides a primer on NVMM programming and NVMM error handling in Linux. Section 3 describes how Pangolin organizes data, manages transactions, and detects and repairs errors. Section 4 presents our evaluations. Section 5 discusses related work. Finally, Section 6 concludes.

2 Background

Pangolin lets programmers build fault-tolerant, crash-consistent data structures in NVMM. This section first introduces NVMM and the DAX mechanism applications use to gain direct access to persistent data. Then, we describe the NVMM error handling mechanisms that Intel processors and Linux provide. Finally, we provide a brief primer on NVMM programming using `libpmemobj` [39], the library on which Pangolin is based.

2.1 Non-volatile Main Memory and DAX

Several technologies are poised to make NVMM common in computer systems. 3D XPoint [30] is the closest to wide deployment. Phase change memory (PCM), resistive RAM (ReRAM), and spin-torque transfer RAM (STT-RAM) are also under active development by memory manufacturers. Flash-backed DRAM is already available and in wide use. Linux and Windows both have support for accessing NVMM and using it as storage media.

The performance and cost parameters of NVMM lie between DRAM and SSD. Its write latency is longer than DRAM, but it will cost less per bit. From the storage perspective, NVMM is faster but more expensive than SSD.

The most efficient way to access NVMM is via direct access (DAX) [15] memory mapping (i.e., DAX-`mmap()`). To use DAX-`mmap()`, applications map pages of a file in an NVMM-aware file system into their address space, so the application can access persistent data from the user-space using load and store instructions, without the file system intervening.

2.2 Handling NVMM Media Errors

To recover from data corruption, Pangolin relies on error detection and media management facilities that the processor and operating system provide together. Below, we describe these facilities available on Intel and Linux platforms. Windows provides similar mechanisms.

Hardware Error Correction Memory controllers for commercially available NVMMs (i.e., battery-backed DRAM and 3D XPoint) implement error-correction code (ECC) in hardware to detect and correct media errors when they can, and they report uncorrectable (but detectable) errors with a machine check exception (MCE) [14] that the operating system can catch and attempt to handle.

Pangolin provides a layer of protection in addition to the ECC hardware provides, but it does not require hardware ECC. Pangolin uses checksums to detect errors that hardware cannot detect. This mechanism also catches software bugs (which are invisible to hardware ECC). ECC does, however, improve performance by transparently handling many media errors.

Regardless of the ECC algorithm hardware provides, field studies of DRAM and SSDs [13, 29, 35, 41, 42, 45] have shown that detectable but uncorrectable media errors occur frequently enough to warrant additional software protection. Furthermore, file systems [23, 49, 51] apply checksums to

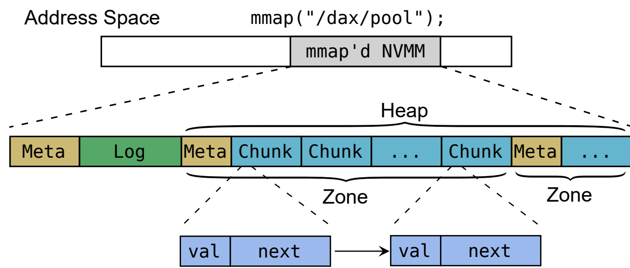


Figure 1: DAX-mapped NVMM as an object store – `Libpmemobj` divides the mapped space into zones and chunks for memory management. The `val` field is a 64-bit integer and the `next` field is a persistent pointer (`PMEMoid`) pointing to the next node object in the pool.

```
1  PMEMobjpool *pool = pmemobj_open("/dax/pool");
2  ...
3  struct node *n = pmemobj_direct(node_oid);
4  n->val = value;
5  pmemobj_persist(pool, &n->val, 8);
6  ...
7  TX_BEGIN(pool) {
8    n = pmemobj_direct(node_oid);
9    pmemobj_tx_add_range(node_oid, 0, sizeof(*n));
10   n->next = pmemobj_tx_alloc(...);
11 } TX_ONABORT {
12   /* handling transaction aborts */
13 } TX_END
14 ...
15 pmemobj_close(pool);
```

Listing 1: A `libpmemobj` program - First modify a node value in a linked list, and later allocate and link a new node from the pool.

their data structures to protect against scribbles.

Repairing Errors When the hardware detects an uncorrectable error, the Linux kernel marks the region surrounding the failed load as "poisoned," and future loads from the region will fail with a bus error. Pangolin assumes an error poisons a 4 KB page since Linux currently manages memory failures at page granularity.

If a running application causes an MCE (by loading from a poisoned page), the kernel sends it a `SIGBUS` and the application can extract the affected address from the data structure describing the signal.

The software can repair the poisoned page by writing new data to the region. In response, the operating system and NVDIMM firmware work together to remap the poisoned addresses to functioning memory cells. The details of this process are part of the Advanced Configuration and Power Interface (ACPI) [46] for NVDIMMs.

Recent kernel patches [6–8, 26] and NVMM library [39] provide utilities for user-space applications to restore lost data by re-writing affected pages with recovered contents (if available).

2.3 NVMM Programming

In this section, we describe `libpmemobj`'s programming model. `Libpmemobj` is a well-supported, open-source C library for programming with DAX-mapped NVMM. It provides facilities for memory management and software transactions that let applications build a persistent object store. Pangolin's interface and implementation are based on `libpmemobj` from PMDK v1.5.

Linux exposes NVMM to the user-space as memory-mapped files (Figure 1). `Libpmemobj` (and Pangolin) refer to the mapped file as a *pool* of persistent objects. Each pool spans a continuous range of virtual addresses.

Within a pool, `libpmemobj` reserves a metadata region that contains information such as the pool's identification (64-bit UUID) and the offset to a "root object" from which all other live objects are reachable. Next, is an area reserved for transaction logs. `Libpmemobj` uses redo logging for its metadata updates and undo logging for application object updates. Transaction logs reside in one of two locations depending on their sizes. Small log entries live in the provisioned "Log" region, as shown in Figure 1. Large ones overflow into the "Heap" storage area.

The rest of the pool is the persistent heap. `Libpmemobj`'s NVMM allocator (a persistent variant of `malloc`/`free`) manages it. The allocator divides the heap's space into several "zones" as shown in Figure 1. A zone contains metadata and a sequence of "chunks." The allocator divides up a chunk for small objects and coalesces adjacent chunks for large objects. By default, a zone is 16 GB, and a chunk is 256 KB.

Listing 1 presents an example to highlight the key concepts of NVMM programming. The code performs two independent operations on a persistent linked list: one is to modify a node's value, and another is to allocate and link a new node.

This example demonstrates two styles of crash-consistent NVMM programming: *atomic*-style (lines 3-5) for a simple modification that is 8 bytes or smaller, and *transactional*-style (lines 7-13) for arbitrary-sized NVMM updates.

Building data structures in NVMM using `libpmemobj` (or any other persistent object library) differs from conventional DRAM programming in several ways:

Memory Allocation `Libpmemobj` provides crash-consistent NVMM allocation and deallocation functions: `pmemobj_tx_alloc`/`pmemobj_tx_free`. They let the programmer specify object type and size to allocate and prevent orphaned regions in the case of poorly-time crashes.

Addressing Scheme Persistent pointers within a pool must remain valid regardless of at what virtual address the pool resides. `Libpmemobj` uses a `PMEMoid` data structure to address an object within a pool. It consists of a 64-bit file ID

and a 64-bit byte offset relative to the start of the file. The `pmemobj_direct()` function translates a `PMEMoid` into a native pointer for use in load or store instructions.

Failure-atomic Updates Modern x86 CPUs only guarantee that 8-byte, aligned stores atomically update NVMM [16]. If applications need larger atomic updates, they must manually construct software transactions. `Libpmemobj` provides undo log-based transactions. The application executes stores to NVMM between the `TX_BEGIN` and `TX_END` macros, and snapshots (`pmemobj_tx_add_range`) a range of object data before modifying it in-place.

Persistence Ordering Intel CPUs provide cache flush/write-back (e.g., `CLFLUSH(OPT)` and `CLWB`) and memory ordering (e.g., `SFENCE`) instructions to make guarantees about when stores become persistent. In Listing 1, the `pmemobj_persist` function and `TX` macros integrate these instructions to flush modified object ranges.

`Libpmemobj` supports a replicated mode that requires a replica pool, doubling the storage the object store requires. `Libpmemobj` applies updates to both pools to keep them synchronized.

Replicated `libpmemobj` can detect and recover from media errors only when the object store is offline, and it cannot detect or recover from data corruption caused by errant stores to NVMM – so-called "scribbles," that might result from a buffer overrun or dereferencing a wild pointer.

3 Pangolin Design

Pangolin allows programmers to build complex, crash-consistent persistent data structures that are also robust in the face of media errors and software "scribbles" that corrupt data. Pangolin satisfies all of the criteria listed in Section 1. This section describes its architecture and highlights the key challenges that Pangolin addresses to meet those requirements. In particular, Pangolin provides the following features unseen in prior works.

- It provides fast, space-efficient recovery from media errors and scribbles.
- It uses checksums to protect object integrity and supports incremental checksum updates.
- It integrates parity and checksum updates into an NVMM transaction system.
- It periodically scrubs data to identify corruption.
- It detects and recovers from media errors and scribbles online.

Pangolin guarantees that it can recover from the loss of any single 4 KB page of data in a pool. In many cases, it can recover from the concurrent loss of multiple pages.

We begin by describing how Pangolin organizes data to protect user objects, library metadata, and transaction logs using a combination of parity, replication, and checksums. Next, we describe micro-buffers and explain how they allow Pangolin to preserve a simple programming interface and protect against software scribbles. Then, we explain how Pangolin detects and prevents NVMM corruption and elaborate on Pangolin's transaction implementation with support for efficient, concurrent updates of object parity. Finally, we discuss how Pangolin restores data integrity after corruption and crashes.

3.1 Pangolin's Data Organization

Pangolin uses replication for its internal metadata and RAID-style parity for user objects to provide redundancy for corruption recovery. The MCE mechanism described in Section 2.2 and object checksums in Pangolin detect corruption.

Pangolin views a zone's chunks as a two-dimensional array as shown in the middle of Figure 2. Each *chunk row* contains multiple, contiguous chunks and the chunks "wrap around" so that the last chunk of a row and the first chunk of the next are adjacent. Pangolin reserves the last chunk row for parity.

In our description of Pangolin, we define a *page column* as a one page-wide, aligned column that cuts across the rows of a zone. A *range column* is similar, but can be arbitrarily wide (no more than a chunk row's size).

Initializing a parity-coded NVMM pool requires zeroing out all the bytes in the file. This is a one-time overhead when creating a pool file and does not affect run-time performance. We report this latency in Section 4.

To detect corruption in user objects, Pangolin adds a 32-bit checksum to the object's header. The header also contains the object's size (64-bit) and type (32-bit). The compiler determines type values according to user-defined object types. Pangolin inherits this design from `libpmemobj` and changes the type identifier from 64-bit to 32-bit for the checksum.

Pangolin's object placement is independent of chunk and row boundaries. Objects can be anywhere within a zone, and they can be of any size (up to the zone size).

In addition to user objects, the library maintains metadata for the pool, zones, and chunks, including allocation bitmaps. Pangolin checksums these data structures to detect corruption and replicates the pool's and zones' metadata for fault tolerance. These structures are small (less than 0.1% for pools larger than 1 GB), so replicating them is not expensive. Pangolin uses zone parity to support recovery of chunk metadata.

Pangolin checksums transaction logs and replicates them for redundancy. It treats log entries in zone storage as zeros during parity calculations. This prevents parity update contention between log entries and user objects (see Section 3.5).

Fault Tolerance Guarantees Pangolin can tolerate a single 4 KB media error anywhere in the pool, regardless of whether it is a data page or a parity page. Based on the bad

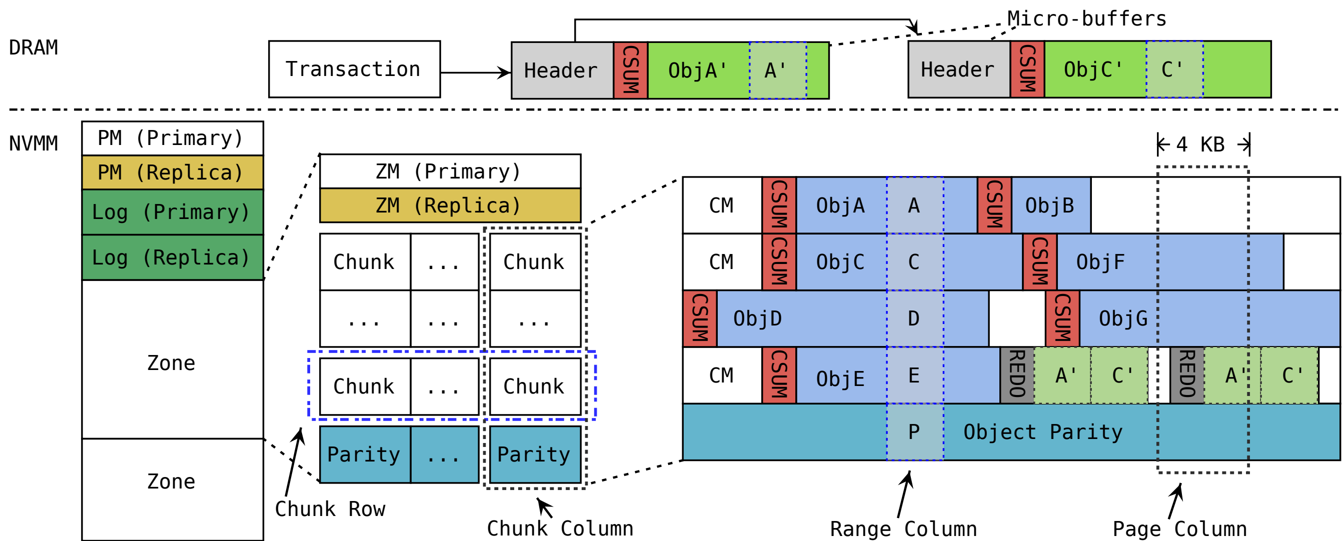


Figure 2: Data protection scheme in Pangolin – Pangolin protects pool metadata (PM), zone metadata (ZM), and chunk metadata (CM). In the highlighted range column, $P = A \oplus C \oplus D \oplus E$. One thread's transaction is modifying ranges A and C of two objects. Pangolin keeps modified data in redo log entries (checksummed and replicated) when the transaction commits. The DRAM part shows micro-buffers for the two objects.

page's address Pangolin can locate its page column and restore its data using other healthy pages.

Faults affecting two pages of the same page column may cause data loss if the corrupted ranges overlap. If an application demands more robust fault tolerance, it can increase the chunk row size, reducing the number of rows and, consequently, the likelihood that two corrupt pages overlap.

Pangolin can recover from scribbles (contiguous overwrites caused by software errors) on NVMM data up to a chunk-row size. By default, Pangolin uses 100 chunk rows, and parity consumes $\sim$1% of a pool's size (e.g., 1 GB for a 100 GB pool).

3.2 Micro-buffering for NVMM Objects

Pangolin introduces micro-buffering to hide the complexity of updating checksums and parity when modifying NVMM objects. Adding checksums to objects and protecting them with parity makes updates more complex, since all three – object data, checksum, and parity – must change at once to preserve consistency. This challenge is especially acute for the atomic programming model as shown in Listing 1 (line 3-5) because a single 8-byte NVMM write cannot host all these updates.

Micro-buffering creates a shadow copy of an NVMM object in DRAM, which separates an object's transient and persistent versions (Figure 2). In Listing 2, `pgl_open` creates a micro-buffer for the node object by allocating a DRAM buffer and copying the node's data from NVMM. It also veri-

```
1  struct node *n = pgl_open(node_oid);
2  n->val = value;
3  pgl_commit(pool, n);
```

Listing 2: A Pangolin transaction for a single-object - This snippet corresponds to line 3-5 of Listing 1.

fies the object's checksum and performs corruption recovery if necessary.

The application can modify the micro-buffered object without concern for its checksum, parity, and crash-consistency because changes exist only in the micro-buffer. When the updates finish, `pgl_commit` starts a transaction that atomically updates the NVMM object, its checksum, and parity (described below). Compared to line 3-5 of Listing 1, Pangolin retains the simple, atomic-style programming model for modifying a single NVMM object, and it supports updates within an object beyond 8 bytes.

Each micro-buffer's header contains information such as its NVMM address, modified ranges, and status flags (e.g., allocated or modified). We elaborate on Pangolin's programming interface and how to construct complex transactions with micro-buffering in Section 3.4.

Another important consideration for micro-buffering is to prevent misbehaving software from corrupting NVMM. If an application's code can directly write to NVMM, as `libpmemobj` allows to, software bugs such as buffer overflows and using dangling pointers can easily cause NVMM

Function	Semantics
`pgl_tx_begin()/commit()/end(), etc.`	Control the lifetime of a Pangolin transaction.
`pgl_tx_alloc()/free()`	Allocate or deallocate an NVMM object.
`pgl_tx_open(PMEMoid oid, ...)`	Create a thread-local micro-buffer for an NVMM object. Verify (and restore) the object integrity, and return a pointer to the micro-buffered user object.
`pgl_tx_add_range(PMEMoid oid, ...)`	Invoke `pgl_tx_open` and then mark a range of the object that will be modified.
`pgl_get(PMEMoid oid)`	Get access to an object, either directly in NVMM or in its micro-buffer, depending on the transaction context. By default, it does not verify the checksum.
`pgl_open(PMEMoid oid, ...)`	Create a micro-buffer for an NVMM object without a transaction. Check the object integrity, and return a pointer to the micro-buffered user object.
`pgl_commit(void *uobj)`	Automatically start a transaction and commit the modified user object in a micro-buffer to NVMM.

Table 1: The Pangolin API - Pangolin's interface mirrors `libpmemobj`'s except that Pangolin does not allow direct writing to NVMM. Pangolin provides single-object transactions using `pgl_open` and `pgl_commit` to convert application code using `libpmemobj`'s atomic updates.

corruption. Conventional debugging tools for memory safety, such as Valgrind [36] and AddressSanitizer [44], insert inaccessible bytes between objects as "redzones" to trap illegal accesses. This approach fails to work for directly accessed NVMM objects because once they are allocated, there is no guarantee for spacing between them, and thus, redzones may land on a nearby, accessible object. One viable approach to using these tools is to let the NVMM allocator insert redzones. However, the presence of redzone bytes will pollute the pool and may exacerbate fragmentation.

Using micro-buffers isolates transient writes from persistent data, and since micro-buffers are dynamically allocated using `malloc()`, they are compatible with existing memory debugging tools. Without using debugging tools, Pangolin also protects micro-buffers by inserting a 64-bit "canary" word in each micro-buffer's header and checks its integrity before writing back to NVMM. On transaction commit, if Pangolin detects a canary mismatch, it aborts the transaction to avoid propagating the corruption to NVMM. Pangolin uses checksums to detect corruptions that may bypass the canary protection.

3.3 Detecting NVMM Corruption

Pangolin uses three mechanisms to detect NVMM corruption. First, it installs a handler for `SIGBUS` (see Section 2.2) that fires when the Linux kernel receives an MCE. A signal handler has access to the address the offending load accessed, and Pangolin can determine what kind of data (i.e., metadata or a user object) lives there and recover appropriately. This mechanism detects media failures, but it cannot discover corrupted data caused by software "scribbles."

To detect scribbles, Pangolin verifies the integrity of user objects using their checksums. Verifying checksums on every access can be expensive. To limit this cost, by default Pangolin only verifies checksums during micro-buffer creation before any object is modified in a transaction. This keeps Pangolin from recalculating a new checksum based on corrupt data. For read-only objects that are accessed by `pgl_get` without micro-buffering, by default Pangolin does not verify checksums. To protect them, Pangolin provides two alternative operation modes: "Scrub" mode runs a scrubbing thread that verifies and restores the whole pool's data integrity when a preset number of transactions have completed, and "Conservative" mode verifies the checksum for every object access (including `pgl_get`). We evaluate the impact of different checksum verification policies in Section 4.

Finally, Linux keeps track of known bad pages of NVMM across reboots. When opening a pool or during its scrubbing, Pangolin can extract this information and recover the data in the reported pages (not currently implemented).

3.4 Fault-Tolerant Transactions

Failure-atomic transactions are central to Pangolin's interface, and they must include verification of data integrity and updates to the checksums and parity data that protect objects. Table 1 summarizes Pangolin's core functions.

Pangolin supports arbitrary-sized transactions and we have made similar APIs and macros as `libpmemobj`'s. The program in Listing 1 can be easily transformed to Pangolin using equivalent functions. One subtle difference is in the handling of atomic-style updates, as shown in Listing 2.

In Pangolin, each thread can execute one transaction or nested transactions (same as `libpmemobj`). Concurrent transactions can execute if each one is associated with a different thread. Currently, Pangolin does not allow concurrent transactions to modify the same NVMM object. Concurrently modifying a shared object may cause data inconsistency if one transaction has to abort. `Libpmemobj` has the same limitation [17].

Each transaction manages its own micro-buffers using a

thread-local hashmap [24], indexed by an NVMM object's `PMEMoid`. Therefore, in a transaction, calling `pgl_tx_open` for the same object either creates or retrieves its micro-buffer. Multiple micro-buffers opened in one transaction form a linked list as shown in Figure 2. Micro-buffers for one transaction are not visible in other transactions, providing isolation.

If a transaction modifies an object, Pangolin copies it to a micro-buffer, performs the changes there, and then propagates the changes to NVMM during commit. Since changes occur in DRAM (which does not require undo information), Pangolin implements redo logging.

At transaction commit, Pangolin recomputes the checksums for modified micro-buffers, creates and replicates redo log entries for the modified parts of the micro-buffers and writes these ranges back to NVMM objects. Then, it updates the affected parity bits (see Section 3.5) and marks the transaction committed. Finally, Pangolin garbage-collects its logs and closes thread-local micro-buffers.

If a transaction aborts, either due to unrecoverable data corruption or other run-time errors, Pangolin discards the transaction's micro-buffers without touching NVMM.

A transaction can also allocate and deallocate objects. Pangolin uses redo logging to record NVMM allocation and free operations, just as `libpmemobj` does.

For read-only workloads, repeatedly creating micro-buffers and verifying object checksums can be very expensive. Therefore, Pangolin provides `pgl_get` to gain direct access to an NVMM object without verifying the object's checksum. The application can verify an object's integrity manually as needed or rely on Pangolin's periodic scrubbing mechanism. Inside a transaction context, `pgl_get` returns a pointer to the object's micro-buffer to preserve isolation.

3.5 Parity and Checksum Updates

Objects in different rows can share the same range of parity, and we say these objects *overlap*. Object overlap leads to a challenge for updating the shared parity because updates from different transactions must serialize but naively locking the whole parity region sacrifices scalability.

For instance, using *ObjA* and *ObjC* in Figure 2, suppose two different transactions modify them, replacing A with A' and C with C', respectively. After both transactions update P, the parity should have the value $P' = A' \oplus C' \oplus D \oplus E$ regardless of how the two transaction commits interleave.

Pangolin uses a combination of two techniques that exploit the commutativity of XOR and fine-grained locking to preserve correctness and scalability.

Atomic parity updates The first approach uses the atomic XOR instruction (analogous to an atomic increment) that modern CPUs provide to perform incremental parity updates for changes to each overlapping object.

In our example, we can compute two parity patches: $\Delta A = A \oplus A'$, $\Delta C = C \oplus C'$ and then rewrite P' as $P \oplus \Delta A \oplus \Delta C$. Since XOR commutes and is a bit-wise operation, the two threads can perform their updates without synchronization.

Hybrid parity updates Atomic XOR is slower than normal or vectorized XOR. For small updates, the latency difference between them is not significant, and Pangolin prefers atomic XOR instructions to update parity without the need for locks. But for large parity updates, atomic XOR can be inefficient. Therefore, Pangolin's hybrid parity scheme switches to vectorized XOR for large transfers.

To coordinate large and small parity updates, Pangolin uses *parity range-locks*, that work similarly as reader/writer locks (or shared mutex): Small writes take shared ownership of a range lock and update parity with atomic XOR instructions. Large updates using vectorized XORs take exclusive ownership of a range-lock, and only one thread can modify parity in a locked range. If one update involves multiple range-locks, serialization happens on a per-range-lock basis.

The managed size of a parity range-lock depends on the performance trade-off between Pangolin's parity mode and `libpmemobj`'s replication mode. We discuss this in Section 4.

Pangolin refreshes an object's checksum in its micro-buffer before updating parity, and it considers the checksum field as one of the modified ranges of the object. Checksums like CRC32 requires recomputing the checksum using the whole object. This can become costly with large objects. Thus, Pangolin uses Adler32 [23], a checksum that allows incremental updates, to make the cost of updating an object's checksum proportional to the size of the modified range rather than the object size.

We implement Pangolin's parity and checksum updates using the Intelligent Storage Acceleration Library (ISA-L) [18], which leverages SIMD instructions of modern CPUs for these data-intensive tasks.

Protections for other transaction systems Other NVMM persistent object systems could apply Pangolin's techniques for parity and checksum updates. For example, consider an undo logging (as opposed to Pangolin's redo logging) system that first stores a "snapshot" copy of an object in the log before modifying the original in-place. In this case, the system could compute a parity patch using the XOR result between the logged data (old) and the object's data (new). Then, it can apply the parity patch using the hybrid method we described in this section.

3.6 Recovering from Faults

In this section, we discuss how Pangolin recovers data integrity from both NVMM corruption and system crashes.

Corruption recovery Pangolin uses the same algorithm to recover from errors regardless of how it detects them (i.e., via `SIGBUS` or a checksum mismatch).

The first step is to pause the current thread's transaction, and to wait until all other outstanding transactions have com-

pleted. Meanwhile, Pangolin prevents the initialization of new transactions by setting the pool's "freeze" flag. This is necessary because, during transaction committing, parity data may be inconsistent.

Once the pool is frozen, Pangolin uses the parity bits and the corresponding parts of each row in the page column to recover the missing data.

Pangolin preserves crash-consistency during repair by making persistent records of the bad pages under recovery. Recovery is idempotent, so it can simply re-execute after a crash.

Pangolin's current implementation only allows one thread to perform any online corruption recovery, and if the thread is executing a transaction, online recovery only works if the thread has not started committing. If two threads encounter faults simultaneously, Pangolin kills the application and performs post-crash recovery (see below) when it restarts. Supporting multi-threaded online recovery, and allowing it to work when threads have partially written NVMM is possible, but it requires complex reasoning about how to restore the data and its parity to a consistent state.

Crash recovery Pangolin handles recovery from a crash using its redo logs. It must also protect against the possibility that the crash occurred during a parity update.

To commit a transaction, Pangolin first ensures its redo logs are persistent and replicated, and then updates the NVMM objects and their parity. If a crash happens before redo logs are complete, Pangolin discards the redo logs on reboot without touching the objects or parity. If redo logs exist, Pangolin replays them to update the objects and then recomputes any affected parity ranges using the data written during replay (which is now known to be correct) and the data from the other rows.

Pangolin does not log parity updates because it would double the cost of logging. This does raise the possibility of data loss if a crash occurs during a parity update and a media error then corrupts data of the same page column before recovery can complete. This scenario requires the simultaneous loss of two pages in the same page column due to corruption and a crash, which we expect to be rare.

4 Evaluation

In this section, we evaluate Pangolin's performance and the overheads it incurs by comparing it to normal `libpmemobj` and its replicated version. We start with our experimental setup and then consider its storage requirements, latency impact, scalability, application-level performance, and corruption recovery.

4.1 Evaluation Setup

We perform our evaluation on a dual-socket platform with Intel's Optane DC Persistent Memory [19]. The CPUs are 24-core engineering samples of the Cascade Lake generation. Each socket has 192 GB DDR4 DRAM and 1.5 TB NVMM. We configure the persistent memory modules in *AppDirect* mode and run experiments on one socket using its local DRAM and NVMM. A recent report [21] studying this platform provides more architectural details.

Pmemobj	`libpmemobj` baseline from PMDK v1.5
Pangolin	Pangolin baseline w/ micro-buffering only
Pangolin-ML	Pangolin + metadata and redo log replication
Pangolin-MLP	Pangolin-ML + object parity
Pangolin-MLPC	Pangolin-MLP + object checksums
Pmemobj-R	`libpmemobj` w/ one replication in another file

Table 2: Library operation modes for evaluation - In the figures, we abbreviate Pangolin as `pgl`.

The CPU provides the `CLWB` instruction for writing-back cache lines to NVMM, non-temporal store instructions to bypass caches, and the `SFENCE` instruction to ensure persistence and memory ordering. It also has atomic XOR and AVX instructions that our parity and checksum computations use.

The evaluation machine runs Fedora 27 with a Linux kernel version 4.13 built from source with the NOVA [48] file system. We run experiments with both Ext4-DAX [27] and NOVA, and applications use `mmap()` to access NVMM-resident files. The performance is similar on the two file systems because DAX-`mmap()` essentially bypasses them.

On our evaluation machine, we found that updating parity with atomic XORs becomes worse than `libpmemobj`'s replication mode when the modified parity range is greater than 8 KB, so we set 8 KB as the threshold to switch between those parity calculation strategies (see Section 3.5).

Table 2 describes the operation modes for our evaluations. The Pangolin baseline implements transactions with micro-buffering. It uses buffer canaries to prevent corruption from affecting NVMM, but it does not have parity or checksum for NVMM data.

We evaluate versions of Pangolin that incrementally add metadata and log replication ("+ML"), object parity ("+MLP"), and checksums ("+MLPC"). We combine the impact of metadata updates with log replication because metadata updates are small and cheap in our evaluation.

Pmemobj-R is the replication mode of `libpmemobj` that mirrors updates to a replica pool during transaction commit. Comparing Pangolin-MLP and Pmemobj-R is especially useful because the two configurations protect against the same types of data corruption: media errors but not scribbles.

4.2 Memory Requirements

We discuss and evaluate Pangolin's memory requirements for both NVMM and DRAM.

NVMM All our Pangolin experiments use a single pool of 100 GB that contains 6× 16 GB zones. Pangolin replicates

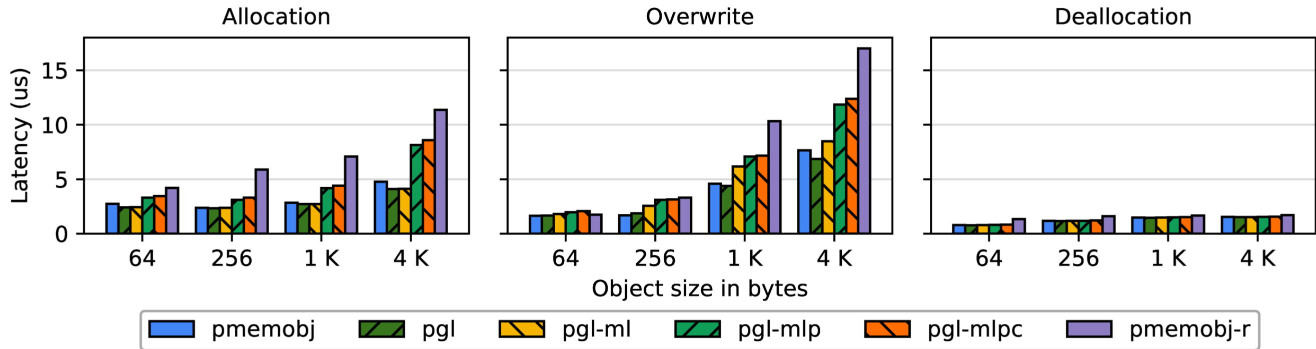


Figure 3: Transaction performance – Each transaction allocates, overwrites, or frees one object of varying sizes. Pangolin's latencies are similar to Pmemobj's. Pangolin-MLP is mostly better than Pmemobj-R because updating parity using atomic XOR and `CLWB` instructions is faster than mirroring data in a separate file.

all the pool's metadata in the same file, which occupies a fixed ~20 MB. The rest of the space is for user objects and their protection data. By default, Pangolin uses 100 chunk rows, so each zone has about 160 MB parity, and that totally occupies ~1% of the pool's capacity. Pmemobj-R uses a second 100 GB file as the replica, doubling the cost of NVMM space requirement.

When using parity, Pangolin has to zero out the whole pool to ensure all zones are initially parity-coded. This takes about 130 seconds. It is a one-time overhead for creating the pool and excluded from the following evaluations.

DRAM Pangolin uses `malloc()`'d DRAM to construct micro-buffers. The required DRAM space is proportional to ongoing transaction sizes. Table 3 summarized the transaction sizes for the evaluated key-value store data structures. Pangolin automatically recycles them on transaction commits. In our evaluation experiments, micro-buffering never exceeds using 50 MB of DRAM.

4.3 Transaction Performance

Figure 3 illustrates the transaction latencies for three basic operations on an NVMM object store: object allocation, overwrite, and deallocation. Each transaction operates on one object, and we vary the size of the object.

For allocation, latency grows with object size for all five configurations, due to constructing the object and cache line write-back latency. Pangolin incurs 2% - 13% lower latencies than Pmemobj due to its use of non-temporal stores for write backs. An allocation operation does not involve object logging, so Pangolin-ML shows performance similar to Pangolin. Pangolin-MLP adds overhead to update the parity data. It outperforms Pmemobj-R by between 1.2× and 1.9×. We found this is because updating parity using atomic XORs and `CLWB`s incurs less latency than mirroring data in a separate file, as Pmemobj-R does.

Adding checksum (Pangolin-MLPC) incurs less than 7% overhead compared to Pangolin-MLP. Parity's impact is larger than checksum's because updating a parity range demands values from three parts: the micro-buffer, the NVMM object, and the old parity data, while computing a checksum only needs data in a DRAM-based micro-buffer. Moreover, Pangolin needs to flush the modified parity range to persistence, which is the same size as the object. In contrary, updating a checksum only writes back a single cache line that contains the checksum value.

Overwriting an NVMM object involves transaction logging for crash consistency. Pangolin and Pmemobj store the same amount of logging data in NVMM, although they use redo logging and undo logging for this purpose, respectively. Since log entry size is proportional to an object's modified size, which is the whole object in this evaluation, this cost grows with the object. With Pangolin, log replication accounts for between 7% to 25% of the latency. Parity updates consume between 8% to 27% of the extra latency, depending on object size, and checksum updates account for less than 5%. Pangolin-MLP's performance for overwrites is 12% worse than Pmemobj-R for 64 B object updates and is between 1.1× and 1.5× better than Pmemobj-R for objects larger than 64 B.

Deallocation transactions only modify metadata, so their latencies do not change much.

4.4 Scalability

Figure 4 measures Pangolin's scalability by randomly overwriting existing NVMM objects and varying the object sizes and the number of concurrent threads.

Pangolin uses reader/writer locks to implement the hybrid parity update scheme described in Section 3.5. The number of rows in a zone and the zone size determine the granularity

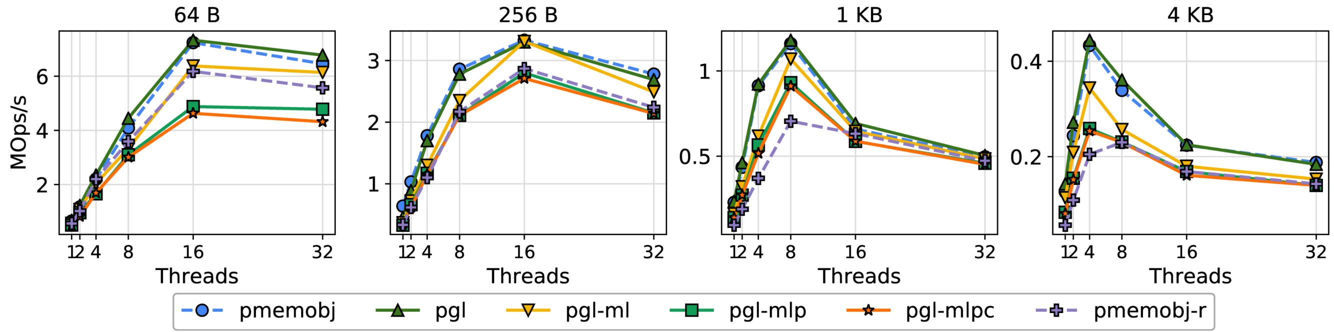


Figure 4: Scalability – Concurrent workloads randomly overwrite objects of varying sizes. Pangolin-MLP scales as well as Pmemobj-R or better for objects larger than 64 B. For 64 B objects, Pangolin-MLP is worse than Pmemobj-R by between 6% and 25% due to synchronization overhead for online recovery.

		ctree	rbtree	btree	skiplist	rtree	hashmap
Object size		56	80	304	408	4136	10 M (table), 40 (entry)
Insert	New	56 (1.00)	80 (1.00)	65.9 (0.22)	408 (1.00)	4502 (1.09)	60.9 (1.00)
	Mod	127.6 (3.28)	330.2 (5.13)	381.2 (1.47)	33.9 (2.50)	200.0 (5.05)	331.1 (4.21)
Remove	New	0	0	0	0	184.1 (0.05)	10.5 (1×10^{-5})
	Mod	28.0 (0.50)	202.8 (2.65)	268.3 (0.90)	16.9 (0.75)	98.6 (2.52)	254.3 (2.16)

Table 3: Data structure and transaction sizes - "Insert" and "Remove" show average transaction sizes for insertions and removals, respectively. "New" and "Mod" indicate average allocated and modified sizes. Value in parentheses is the average number of objects involved in the transaction.

of these locks: For a fixed zone size, more rows means fewer columns and fewer parity range-locks.

There is no lock contention in the results because the transactions use atomic XOR instructions and can execute concurrently (only taking the reader locks). Our configuration with 1% parity (160 MB parity per 16 GB zone) has 20 K range-locks per zone, so the chance of lock contention is slim even with large updates (more than 8 KB) and many cores.

The graphs also show how each Pangolin's fault-tolerance mechanisms affect performance. Pangolin's throughput is very close to Pmemobj. Pangolin-MLP mostly outperforms Pmemobj-R for object updates that are 256 B or larger, up to 1.5×. But for 64 B object updates, it performs worse than Pmemobj-R by between 6% and 25%. This is because when enabling parity, every Pangolin transaction checks the pool freeze flag (an atomic variable), incurring synchronization overhead. This overhead is noticeable for short transactions with 64 B objects but becomes negligible for larger updates. Pangolin-MLPC only performs marginally worse than Pangolin-MLP.

Scaling degrades for all configuration as update size and thread count grow because the sustainable bandwidth of the persistent memory modules becomes saturated.

4.5 Impacts on NVMM Applications

To evaluate Pangolin in more complex applications, we use six data structures included in the PMDK toolkit: crit-bit tree (ctree), red-black tree (rbtree), btree, skiplist, radix tree (rtree), and hashmap. They have a wide range of object sizes and use a diverse set of algorithms to insert, remove, and lookup values. We rewrite these benchmarks with Pangolin's programming interface as described in Section 3.4.

Table 3 summarizes the object and transaction sizes for each workload. The tree structures and the skiplist have a single type of object, which is the tree or list node. Hashmap has two kinds of objects. One is the hash table that contains pointers to buckets. The hash table grows as the application inserts more key-value pairs. Each bucket is a linked list of fixed-sized entry objects.

Each insertion or removal is a transaction processing a key-value pair. The workloads involve a mix of object allocations, overwrites, and deallocations. Table 3 shows, on average, the number of bytes and objects (in parentheses) involved in each data structure's transaction. Deallocated sizes are not shown because they marginally affect the performance differences (see Figure 3).

An average allocation size ("New" rows in the table) smaller than the object size means the data structure does not allocate a new object for every insert operation (e.g., btree).

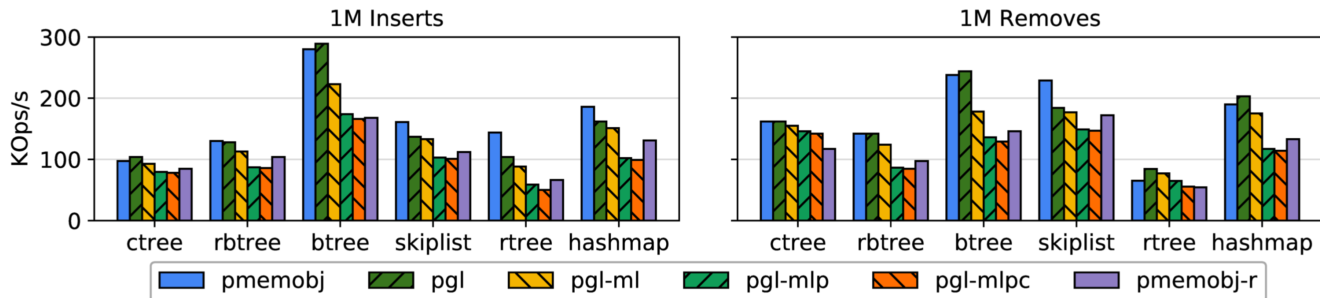


Figure 5: Key-value store performance – Each transaction either inserts or removes one key-value pair from the data store. Pangolin performs similarly to Pmemobj except for cases when a transaction's modified size is much less than an object's size (e.g., skiplist and rtree) due to micro-buffering overhead. Pangolin-MLP's performance is 95% of Pmemobj-R on average.

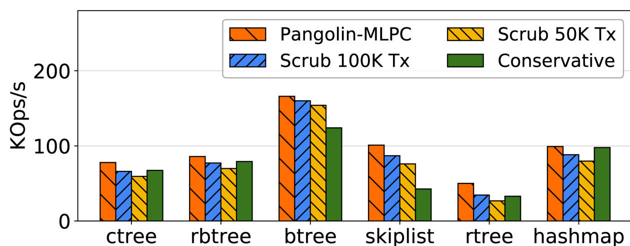


Figure 6: Checksum verification impact – Pangolin-MLPC bars are the same as those in Figure 5 for 1M Inserts. The cost of different policies depends strongly on data structures.

	ctree	rbtree	btree	sklist	rtree	hmap
Pmemobj	1.0	1.0	1.0	1.0	1.0	1.0
Pgl-MLPC	0.92	0.84	0.87	0.96	0.42	0.42
Scrub 100K	0.10	0.09	0.09	0.10	0.04	0.05
Scrub 50K	0.05	0.04	0.05	0.05	0.02	0.02
Conservative	0	0	0	0	0	0

Table 4: Vulnerability evaluation - Each row shows object bytes (normalized to Pmemobj) accessed without checksum verification.

The average modified sizes ("Mod" rows) determine the logging size and affect the performance drop between Pangolin and Pangolin-ML. Note that a transaction does not necessarily modify (and log) the whole range of an involved object. The performance difference between Pangolin-ML and Pangolin-MLP is a consequence of both allocated and modified sizes.

For insert transactions, Pangolin is faster than Pmemobj for ctree and btree, but slower than Pmemobj for other data structures. This is because the slower applications have relatively small modified sizes compared to the object sizes, and Pangolin's data movement from NVMM to micro-buffers overshadows its advantage for whole-object updates, as shown in Figure 3. For remove transactions, Pangolin is marginally faster than Pmemobj except for the case of skiplist, which is also because of the data movement caused by micro-buffering.

Pangolin-MLP's performance is 95% of Pmemobj-R on average, and it saves orders of magnitude NVMM space by using parity data as redundancy. Pangolin-MLPC adds scribble detection and performance drops by between 1.5% to 15% relative to Pangolin-MLP. Adding object checksums impacts rtree's transactions the most because the allocated object size is large, which requires more checksum computing time.

Pangolin does not impact the lookup performance because it performs direct NVMM reads without constantly verifying object checksums. Pangolin ensures data integrity with its checksum verification policy, as discussed in Section 3.3.

Figure 6 illustrates the impact of different strategies for checksum verification. We compare Pangolin's default mode (Pangolin-MLPC) with two "Scrub" modes and a "Conservative" mode. The default mode only verifies checksums for micro-buffered objects. In "Scrub" mode, a scrubbing thread verifies data integrity of the whole object pool when a preset number (indicated by legends in Figure 6) of transactions have completed. The "Conservative" mode verifies the checksum for every object access (including those read by `pgl_get` without micro-buffering).

Table 4 quantifies the vulnerability using the amount of object data that is accessed without checksum verification. The data accumulates across all transactions for Pmemobj, Pangolin-MLPC, and "Conservative" modes. For "Scrub" modes, we count the vulnerable data between two scrubbing runs. Numbers in Table 4 are normalized to Pmemobj, which does not have any checksum protection for object data.

The cost of verifying checksums for every object access depends strongly on the data structure size and its insertion algorithm. For small objects, such as ctree, rbtree, and hashmap, the cost is negligible. But for btree, skiplist, and rtree, due to their large object sizes, the cost is significant. Thus, a scrubbing-based policy could be faster, with more data subject

to corruption between two successive runs.

4.6 Error Detection and Correction

Pangolin provides error injection functions to emulate both hardware-uncorrectable NVMM media errors and hardware-undetectable scribbles.

We initially developed Pangolin using conventional DRAM machines that lack support for injecting NVMM errors at the hardware level. Therefore, we use `mprotect()` and `SIGSEGV` to emulate NVMM media errors and `SIGBUS`. When an NVMM file is DAX-mapped, the injector can randomly choose a page that contains user-allocated objects, erase it, and call `mprotect(PROT_NONE)` on the page. Later, when the application reads the corrupted page, Pangolin intercepts `SIGSEGV`, changes the page to read/write mode, and restores the page's data. The injector function can also randomly corrupt a metadata region or a victim object to emulate software-induced, scribble errors.

In both test cases, we observe Pangolin can successfully repair a victim page or an object and resume normal program execution. In our evaluation using a 100 GB pool and 1 GB parity, we measured 180 μs to repair a page of a page column.

We also intentionally introduce buffer overrun bugs in our applications and observe that Pangolin can successfully detect them using micro-buffer canaries. The transaction then aborts to prevent any NVMM corruption. We have also verified Pangolin is compatible with AddressSanitizer for detecting buffer overrun bugs (when updating a micro-buffered object exceeds its buffer's boundary), if both Pangolin and its application code are compiled with support.

5 Related Work

In this section, we place Pangolin in context relative to previous projects that have explored how to use NVMM effectively.

Transaction Support All previous libraries for using NVMMs to build complex objects rely on transactions for crash consistency. Although we built Pangolin on `libpmemobj`, its techniques could be applied to another persistent object system. NV-Heaps [3], Atlas [1], DCT [22], and `libpmemobj` [39] provide undo logging for applications to snapshot persistent objects before making in-place updates. Mnemosyne [47], SoftWrAp [11], and DUDETM [25] use variations of redo logging. REWIND [2] implements both undo and redo logging for fine-grained, high-concurrent transactions. Log-structured NVMM [12] makes changes to objects via append-only logs, and it does not require extra logging for consistency. Romulus [5] uses a main-back mechanism to implement efficient redo log-based transactions.

None of these systems provide fault tolerance for NVMM errors. We believe they can adopt Pangolin's parity and checksum design to improve their resilience to NVMM errors at low storage overhead. In Section 3.5 we described how to apply the hybrid parity updating scheme to an undo logging-based system. Log-structured and copy-on-write systems can adopt the techniques in similar ways.

Fault Tolerance Both Pangolin and `libpmemobj`'s replication mode protect against media errors, but Pangolin provides stronger protection and much lower space overhead. Furthermore, `libpmemobj` can only repair media errors offline, and it does not detect or repair software corruption to user objects.

NVMalloc [32] uses checksums to protect metadata. It does not specify whether application data is also checksum-protected, and it does not provide any form of redundancy to repair the corruption. NVMalloc uses `mprotect()` to protect NVMM pages while they are not mapped for writing. Pangolin could adopt this technique to prevent an application from scribbling its own persistent data structures.

The NOVA file system [48, 49] uses parity-based protection for file data. However, it must disable these features for NVMM pages that are DAX-mapped for writing in user-space, since the page's contents can change without the file system's knowledge, making it impossible for NOVA to keep the parity information consistent if an application modifies DAX-mapped data. As a result, Pangolin's and NOVA's fault tolerance mechanisms are complementary.

6 Conclusion

This work presents Pangolin, a fault-tolerant, DAX-mapped NVMM programming library for applications to build complex data structures in NVMM. Pangolin uses a novel, space-efficient layout of data and parity to protect arbitrary-sized NVMM objects combined with per-object checksums to detect corruption. To maintain high performance, Pangolin uses micro-buffering, carefully-chosen parity and checksum updating algorithms. As a result, Pangolin provides stronger protection, better availability, and much lower storage overhead than existing NVMM programming libraries.

Acknowledgments

This work was supported in part by Toshiba, and we are thankful for their continued sponsorship. We thank our shepherd, Kimberly Keeton and the anonymous USENIX ATC reviewers for their insightful comments and suggestions. We are also thankful for Intel to provide us early access to platforms with Intel's Optane DC Persistent Memory Modules.

References

[1] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM*

International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'14, pages 433–452. ACM, 2014.

[2] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. REWIND: Recovery Write-ahead System for In-Memory Non-volatile Data-Structures. *Proceedings of the VLDB Endowment*, 8:497–508, 2015.

[3] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 105–118. ACM, 2011.

[4] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, SOSP'09, pages 133–146. ACM, 2009.

[5] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA'18, pages 271–282. ACM, 2018.

[6] Dan Williams. libnvdimm for 4.12, 2017. https://lkml.org/lkml/2017/5/5/620.

[7] Dan Williams. libnvdimm for 4.13, 2017. https://lkml.org/lkml/2017/7/6/843.

[8] Dan Williams. use memcpy_mcsafe() for copy_to_iter(), 2018. https://lkml.org/lkml/2018/5/1/708.

[9] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'16, pages 125–136. ACM, 2016.

[10] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[11] Ellis R Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, MSST'16, pages 1–14. IEEE, 2015.

[12] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-Structured Non-Volatile Main Memory. In *Proceedings of the USENIX Annual Technical Conference*, ATC'17, pages 703–717. USENIX Association, 2017.

[13] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 111–122. ACM, 2012.

[14] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/en-us/isa-extensions.

[15] Intel. Introduction to Programming with Persistent Memory from Intel, 2017. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent\-memory-from-intel.

[16] Intel. Persistent Memory Programming - Frequently Asked Questions, 2017. https://software.intel.com/en-us/articles/persistent-memory-programming-\frequently-asked-questions.

[17] Intel. Discover Persistent Memory Programming Errors with Pmemcheck, 2018. https://software.intel.com/en-us/articles/discover-\persistent-memory-programming-errors-with-\pmemcheck.

[18] Intel. Intelligent Storage Acceleration Library, 2018. https://software.intel.com/en-us/storage/isa-l.

[19] Intel. Intel® Optane™ DC persistent memory, 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 427–442. ACM, 2016.

[21] 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, 2019. https://arxiv.org/abs/1903.05714.

[22] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 399–411. ACM, 2016.

[23] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 197–212. USENIX Association, 2017.

[24] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 27:1–27:14. ACM, 2014.

[25] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 329–343. ACM, 2017.

[26] Tony Luck. Patchwork mm/hwpoison: Clear PRESENT Bit for Kernel 1:1 Mappings of Poison Pages, 2017. https://patchwork.kernel.org/patch/9793701.

[27] LWN. Add Support for NV-DIMMs to Ext4, 2014. https://lwn.net/Articles/613384.

[28] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys'17, pages 499–512. ACM, 2017.

[29] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'15, pages 177–190. ACM, 2015.

[30] Micron. 3D XPoint Technology, 2017. http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[31] Micron. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility, 2017. http://www.micron.com/products/dram-modules/nvdimm.

[32] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS'13. ACM, 2013.

[33] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 135–148. ACM, 2017.

[34] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 401–410. ACM, 2012.

[35] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR'16, pages 7:1–7:11. ACM, 2016.

[36] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100. ACM, 2007.

[37] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, HPCA'18, pages 336–349. IEEE, 2018.

[38] Christian Perone and David Murray. pynvm: Non-volatile memory for Python, 2017. https://github.com/pmem/pynvm.

[39] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[40] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'15, pages 672–685. IEEE, 2015.

[41] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, volume 6, pages 9:1–9:23, New York, NY, USA, September 2010. ACM.

[42] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'09, pages 193–204. ACM, 2009.

[43] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 91–104. ACM, 2017.

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, ATC'12. USENIX Association, 2012.

[45] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 297–310. ACM, 2015.

[46] UEFI Forum. Advanced configuration and power interface specification, 2017. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 91–104. ACM, 2011.

[48] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 323–338. USENIX Association, 2016.

[49] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 478–496. ACM, 2017.

[50] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.

[51] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10. USENIX Association, 2010.

Pisces: A Scalable and Efficient Persistent Transactional Memory

Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang,
Binyu Zang, Haibing Guan, Haibo Chen
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University

Abstract

Persistent transactional memory (PTM) programming model has recently been exploited to provide crash-consistent transactional interfaces to ease programming atop NVM. However, existing PTM designs either incur high reader-side overhead due to blocking or long delay in the writer side (efficiency), or place excessive constraints on persistent ordering (scalability).

This paper presents Pisces, a read-friendly PTM that exploits snapshot isolation (SI) on NVM. The key design of Pisces is based on two observations: the redo logs of transactions can be reused as newer versions for the data, and an intuitive MVCC-based design has read deficiency. Based on the observations, we propose a dual-version concurrency control (DVCC) protocol that maintains up to two versions in NVM-backed storage hierarchy. Together with a three-stage commit protocol, Pisces ensures SI and allows more transactions to commit and persist simultaneously. Most importantly, it promises a desired feature: hiding NVM persistence overhead from reads and allowing nearly non-blocking reads.

Experimental evaluation on an Intel 40-thread (20-core) machine with real NVM equipped shows that Pisces outperforms the state-of-the-art design (i.e., DUDETM) by up to 6.3× for micro-benchmarks and 4.6× for TPC-C new order transaction, and also scales much better. The persistency cost is from 19% to 50% for 40 threads.

1 Introduction

Non-volatile memory (NVM) such as phase-change memory (PCM) [46, 67, 78], resistive random-access memory (ReRAM) [9, 45], and Intel/Micron's 3D-XPoint [2, 4], is revolutionizing the storage hierarchy thanks to the promising features like byte-addressability and non-volatility with a close-to-DRAM speed. By supporting persistent data access via CPU load/store instructions, these technologies bring ample opportunities for applications to achieve optimal performance as well as efficient crash consistency [20, 62].

To efficiently program on NVM with a balance among good programmability, high performance, and low software overhead, persistent transactional memory (PTM), also known as durable (memory) transactions, has been exploited by prior work [21, 37, 44, 49, 56, 74, 76]. Through combining transactional memory [29, 34, 66, 68, 70] and NVM, PTM offers the properties of atomicity, consistency, isolation, durability (ACID) to applications on NVM.

To ensure the durability for transactions, some prior designs [21, 44, 56, 74] need to persist a transaction's log while holding the locks of the data being modified or explicitly track the dependencies among transactions through locks. This, however, may block concurrent read operations on the same data. The long blocking duration may become a severe performance bottleneck due to the amplified persistence overhead incurred by high write latency of NVM (usually 10× compared to DRAM) (*low read efficiency*). This is especially true when read operations dominate in many workloads [15, 19, 52, 63]. In contrast, another design [49] eliminates the persistence latency from a transaction's critical path through relaxing the durability semantics, i.e., making a transaction's modifications visible before its log reaches NVM. However, such a design sacrifices the durability guarantee and requires to apply logs back to the durable data according to a total order. Unfortunately, such a strict persistence ordering may be the bottleneck of scalability since it is hard to parallelize the persistence operations. Overall, it is challenging to design a PTM system that insulates readers from being affected by high NVM persistence overhead while enforcing strong durability as well as avoids overly-constrained persistence ordering simultaneously.

We notice that *snapshot isolation* (SI) [8, 12] can avoid read-write conflicts and suffices for many real-world applications [6, 11, 26, 28, 33, 47, 48, 61, 68, 77], which makes it possible to design a PTM that allows a transaction to persist its log in its critical path (no sacrifice the durability), while hiding the high persistence overhead from concurrent read operations. Multi-version concurrency control (MVCC) [13, 28] is a common choice to achieve SI. We *observe* that the (redo) logs which will be finally applied to the durable data (old) can be regarded as a new data version, which enables us to efficiently introduce MVCC to PTM.

However, after a deep analysis, we *find* that a straightfor-

ward MVCC-based PTM design not only brings high reader-side overhead due to read-indirection problems (challenge-1), i.e., locating the consistent objects in the version lists, but also still leaves the readers affected by the NVM persistence overhead (challenge-2). So, we further present *Pisces*, a read-friendly PTM design that also embraces SI while solving the above two problems and achieves both high read efficiency and good scalability.

Specifically, Pisces proposes *dual-version concurrency control* (*DVCC*) inspired by MVCC and scalable synchronization primitives [18, 51, 54, 55], to solve challenge-1. DVCC still avoids read-write conflicts and thus allows high parallelism for transaction execution, but only keeps one or two versions (using the log as the newer version) for each data object, which minimizes the high cost for maintaining multiple versions in NVM as well as searching in the version lists. To solve challenge-2, Pisces hides the NVM persistence overhead from readers through *three-stage commit* that separates the durable point and the visible point of a (read-write) transaction and minimizes the possible read-blocking time to the duration of two DRAM stores. This blocking *rarely* happens since the possible blocking time is extremely short. A transaction persists its logs (new versions for objects) into NVM in a persist stage (durable) and makes its logs readable to other transactions in a following concurrency commit stage (visible). Note that the potential blocking period resides in the latter stage. Hence, the NVM persistence overhead can be hidden from readers. Besides, a transaction eagerly reclaims the old versions of objects and overrides them with new versions in the last write-back stage, which is for avoiding indirect reads. Furthermore, Pisces also enables *flush-diff* (persist modifications only) to prevent excessive NVM persistence operations and leverages *group-commit* to reduce the overhead for write transactions.

In all, Pisces hides the NVM persistence overhead from readers and promises *almost non-blocking* reads. Pisces guarantees *snapshot isolation* (a formal proof is also provided), and promises *crash consistency* that can restore the system to a consistent snapshot after crashes. In essence, Pisces explores a trade-off between isolation and performance by loosening the isolation level for better performance.

We have implemented and evaluated Pisces on a 40-thread machine. Evaluation results show that Pisces has a notably higher throughput and better scalability compared with the state-of-the-art design (i.e., DUDETM [49]). Specifically, Pisces achieves up to 6.3× throughput improvement in micro-benchmarks and can improve the throughput of TPC-C new order transaction [24] and TATP benchmark [72] by 460% and 64%, respectively.

In summary, this paper makes the following contributions:

- An observation that redo logs can be used as newer data versions and an intuitive MVCC-based PTM design with the observation. A careful analysis of the read-inefficiency of the MVCC-based design.
- A first PTM with snapshot isolation (Pisces), which leverages DVCC and three-stage commit to benefit readers most.
- An implementation and evaluation on a real machine with NVM that demonstrate Pisces's efficiency and scalability.

2 Background & Overview

Comparing with database transaction, Transactional memory (TM) [38] ensures atomicity, isolation and consistency (ACI), but lacks the important property of durability. However, the emergence of non-volatile memory provides an opportunity to equip TM with durability [44, 49, 56, 74, 76]. This section first introduces the backgrounds of NVM and PTM, then provides an overview of our system.

2.1 Background

NVM. The recent release of Intel Optane DC Persistent Memory [2] marks the transition of non-volatile memory (NVM) technology from research prototypes to mainstream products. NVM promises to provide fast data persistency. According to current studies [47, 79, 81], NVM has the following three features. First, most NVM designs are byte-addressable. This is one major reason why we can directly replace DRAM with NVM. Second, NVM has close-to-DRAM read latency, but about 10× write latency comparing with DRAM. For example, PCM's write latency is 150∼1000ns and ReRAM's is 500ns, while DRAM has only 60ns write latency [47, 79]. Third, special instructions [3] are provided to help persist the data: 1). *pflush* (e.g., *clwb*) will flush a cache line from CPU cache to NVM. 2). *pfence* (e.g., *mfence*) ensures all previous *pflush* instructions finish.

PTM. There are already various researches which build Persistent Transactional Memory (PTM) systems by leveraging NVM [21, 23, 44, 49, 56, 74]. However, most of them focus on optimizing the persistence overhead [21, 44, 56, 74]. For example, Kamino-tx [56] removes the overhead of data copy in a transaction's critical path by maintaining a backup of all the data. However, in these systems, an on-going write operations usually block conflicting read operations to ensure the consistency. As a result, these read operations will also suffer from NVM's high write latency.

A state-of-the-art design, called DUDETM [49], tries to address this issue with a decoupled PTM design: it temporarily buffers the running transaction's updates and its redo log in DRAM, then a number of threads will flush the log to NVM asynchronously. Furthermore, a dedicated thread (named reproduce thread) will replay the log to apply the updates to the persistent objects in NVM. However, once the log buffer becomes full because the reproduce thread cannot timely replay and clean the logs, the system needs to stall to wait for the reproduce thread to catch up. To ensure the consistency of the persistent state, the reproduce thread needs to replay the operations in the log sequentially. As a result,

the reproduce thread harms the system scalability. Figure 1-(a) shows the scalability issue of DUDETM: its performance can only scale up to 8 cores, after which the performance will be bottlenecked by the reproduce thread.

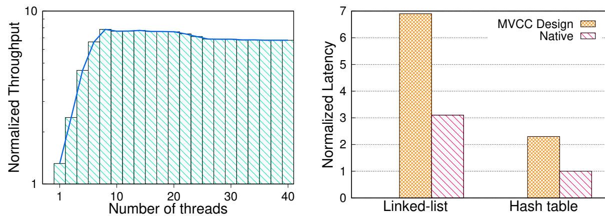


Fig. 1: (a) A hash table benchmark with 40% update rate. (b) A comparison on read-only transaction latency.

2.2 Overview

Goal. Comparing with existing works, Pisces is a PTM system with a read-friendly design, since lots of workloads using transactions are read-dominated [15, 19, 22, 30, 52, 63]: for example, the read-write ratio in the update operation of an 8 layer (8-15 keys per node) B+-tree is about 80:1; In the TPC-E and TATP [25, 72] benchmarks, about 80% of the transactions are read-only.

Strawman. Pisces is based on the intuition that *snapshot isolation* (SI) [8, 12] is able to avoid blocking reads by conflicting writes. At the same time, SI is applicable not only to database workloads [6, 11, 26, 28, 33, 47, 77], but also to TM workloads [48, 61, 68]. For example, Lu et al. [53, 68] prove that SI is enough to support a concurrent skip list. Thus, both database and STM systems have begun to use SI to improve concurrency [48, 61, 68].

However, is an intuitive SI implementation good enough to achieve our goal, a read oriented PTM? To answer this question, we implement a prototype system to provide SI based on multi-version concurrency control. Each object maintains a list of multiple versions and is identified by an ID. Each version has a timestamp to indicate its committed point-in-time. When a transaction starts, it sets a start timestamp based on the global timestamp kept by the system. During execution, to read an object, a transaction finds the most recent version which has a smaller timestamp than the transaction's start timestamp by traversing the object's list. For write, a transaction buffers updates in the write set and records the operation in a redo log. To commit its updates, a transaction first acquires the locks of all objects it tries to update. Then it detects write-write conflicts by checking the latest timestamp of these objects. If any object's latest timestamp is larger than the transaction's start timestamp, then the transaction is aborted. After passing the validation, the transaction retrieves its commit timestamp and updates the global timestamp. Then, the transaction flushes its log and the commit timestamp from CPU cache into the NVM, then flushes the updates in the write set to the persistent objects in NVM. At last, it releases all locks.

Issues. To analyze the efficiency of this design, we use it to implement concurrent data structures and compare with their native (single-threaded) implementations. Figure 1-(b) shows the evaluation results: the intuitive design has considerable overhead on read requests' latency because of the following problems:

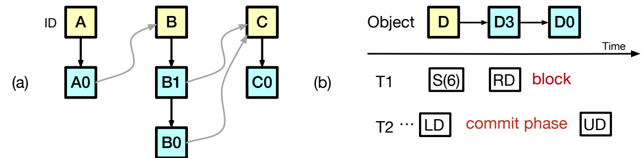


Fig. 2: (a) An ordered linked list structure in our MVCC-based PTM. Black arrows represent pointers in different version lists, while the gray arrows indicate the pointers in the linked list. (b) Read operations get blocked in MVCC.

First, traversing the multi-version list of each object increases the latency. This is not only because a transaction with small start timestamp may need to perform multiple indirect memory accesses, but also because the random accesses may harm the cache locality and get blocked due to the cache line being evicted to NVM. Figure 2 (b) gives a simple example: if a transaction traverses this 3-object list, it actually needs to traverse a much longer list which at least contains three objects and three object IDs.

Second, read operations may still be blocked by the NVM persist operations. Specifically, when a reader accesses an object which is locked by a writer, the reader may be blocked until the writer commits. The reason is the reader is not sure if the writer will have a smaller commit timestamp than its start timestamp or not. Unfortunately, the writer cannot commit until it flushes all its logs into NVM and applies the updates in its write set. Figure 2 (b) gives a simple example: T_1 starts with timestamp 6. When it reads object D, it finds D is locked by T_2. Then, it has to be blocked, as T_2 may update D with a timestamp smaller than 6.

Basic idea. We develop *Pisces* to solve above issues based on the following basic designs:

Dual-version concurrency control (DVCC). To reduce the cost of traversing an object's list, Pisces keeps up to two versions for each object: original object and object copy. When a transaction tries to write an object, it creates and links a new copy to the original object. When the transaction commits, it writes the object copy back to the original object. Concurrent transactions update the same object exclusively by acquiring a lock. Read transactions are able to directly access either version based on their timestamps. Furthermore, to reduce unnecessary NVM writes, we reuse the updates in the redo log as object copy. However, the challenge to implement DVCC is how to ensure an original object won't be overwritten when it may still be needed by some outstanding

transactions with smaller start timestamps.

Three-stage commit protocol. To reduce the blocking overhead in the MVCC design, Pisces proposes a three-stage commit protocol: the commit phase is divided into concurrency commit stage, and write-back stage. In the persist stage, a transaction flushes its log into NVM. In the concurrency commit stage, the transaction updates its end timestamp (commit timestamp) and the timestamps of all the object copies in the redo log atomically. In the write-back stage, the transaction writes all object copies back to their original objects. By decoupling different functionalities of the commit phase, Pisces allows nearly non-blocking reads. But the challenge lies in how to atomically update both the end timestamp and the timestamps of object copies efficiently.

Limitation. The main limitation of Pisces is it only provides SI which does not work for all applications, and SI suffers from the well-known write skew anomaly[1] under certain conditions. However, there is a long line of research [16, 33, 48, 53] on how to detect or eliminate write skew anomalies for SI. Moreover, making SI serializable is also well studied [59, 64, 69, 75]. Leveraging these techniques to provide a stronger isolation level is future work. Currently, careful programming on Pisces is required.

3 Design

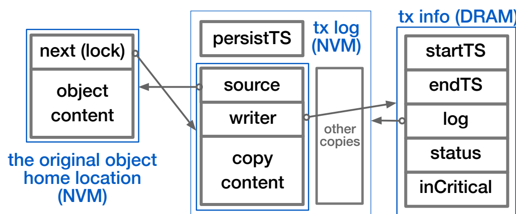


Fig. 3: The memory layout of an object, per-transaction log and per-thread metadata. Arrows represent pointers.

Layout. Figure 3 shows the memory layout of three critical components (data object, per-transaction log and per-thread metadata) in Pisces. Pisces attaches each object with a pointer (named *next* and initialized as 0) which may point to a next version of this object and is also used as a write lock that needs to be exclusively acquired by a writer. Pisces pre-allocates log area for each thread and each transaction gets its log from the log area of the execution thread when it begins. Pisces also keeps per-thread metadata to record the metadata of the running transaction in each thread (a thread executes at most one running transaction at a time). Object copies as the next (newer) versions of objects reside in the transactions' log and each object copy contains two pointers. One is named as *source* and points to the original object. The other is named as *writer* and points to the running transaction that owns this log (creates this copy).

[1] A typical write skew example is: One transaction reads A and writes B while another concurrent transaction reads B and writes A.

For the challenges mentioned in the above section, we provide simple but efficient solutions accordingly: First, to prevent an original version from being falsely overwritten, we leverage an RCU-similar design (grace period detection) to block the writer in the write-back phase until the original version in home location is safe to be overwritten. Note that the blocking time will not be exposed to readers. Second, to atomically update a transaction's end timestamp and each copy's version timestamp in an efficient way, we do not explicitly maintain the version timestamp for each copy. Instead, each copy contains the *writer* pointer and reuses the transaction's end timestamp as its version timestamp. As a result, atomicity is guaranteed by simply updating the end timestamp of the write transaction. Next, we discuss the details about the algorithm whose pseudo code is provided in Algorithm 1 and the correctness argument.

3.1 Algorithm

TM_Start begins a transaction. A transaction marks its status as ACTIVE first, executes a *fence* instruction and reads the global timestamp (globalTS) as its start timestamp (startTS). The *fence* instruction ensures line 2 is executed before line 4.

TM_Read returns a pointer for reading an object. It first reads the value of next pointer in the original object and returns the original object directly if next is zero. This is the *fast path*: accessing the pointer located just before the object introduces nearly-zero overhead because the CPU will prefetch adjacent cache lines. If next is non-zero, which means there exists an object copy, *TM_Read* returns the object copy when it is created by the current transaction (line 12) or its version is no greater than the current transaction's start timestamp (line 15). There is only one rare case in which *TM_Read* needs to wait (line 14). We discuss this later when introducing *TM_Commit*.

TM_Write returns a pointer for writing an object. A transaction can directly write a copy created by itself (line 21 to 23). When writing an object for the first time, a transaction reserves an area for the object copy in its log and tries to acquire the object's write lock (i.e., next pointer) with a compare-and-swap instruction (line 25). If fails to lock, the transaction aborts and restarts after a random delay, which also avoids deadlocks. Otherwise, it copies the original object's content to the object copy and can directly read or write the copy now. Pisces makes copies (redo log) at object granularity, which mitigates the *read-indirection problem* of redo logging at byte granularity. Pisces chooses encounter-time locking to detect conflicts early and thus can avoid unnecessary NVM writes. Also, an object's write lock ensures that it can only be updated sequentially.

TM_Commit always successfully commits a transaction. A transaction marks its status as INACTIVE, indicating it no longer reads any object. It commits directly if it is a read-only transaction. A read-write transaction needs to go through

Algorithm 1: Pseudo code of Pisces

```
1: Function TM_START(tx)
2:   tx.status = ACTIVE
3:   fence
4:   tx.startTS = globalTS
5:   tx.endTS = INF
6:
7: Function TM_READ(tx, p_obj)
8:   next = p_obj.next
9:   if next is EMPTY then
10:    return p_obj          // fast path
11:  wtx = next.writer
12:  if wtx is tx then
13:    return next
14:  wait until wtx.inCritical is FALSE
15:  if wtx.endTS ≤ tx.startTS then
16:    return next
17:  else
18:    return p_obj
19:
20: Function TM_WRITE(tx, p_obj)
21:  if p_obj.next is NON-EMPTY
22:  and p_obj.next.writer is tx then
23:    return p_obj.next
24:  copy = tx.log.alloc(p_obj)
25:  copy.writer = tx
26:  if CAS(p_obj.next, EMPTY, copy) fails then
27:    abort()
28:  copy.source = p_obj
29:  memcpy_content(copy, p_obj)  // p_obj -> copy
30:  return copy
31:
32: Function TM_COMMIT(tx)
33:  tx.status = INACTIVE
34:  // stage 1: persist stage
35:  if tx.log is EMPTY then
36:    return
37:  pflush(tx.log)
38:  pfence
39:  tx.log.persistTS = globalTS
40:  pflush(tx.log.persistTS)
41:  pfence
42:  // stage 2: concurrency commit stage
43:  tx.inCritical = TRUE
44:  fence
45:  tx.endTS = globalTS + 1
46:  tx.inCritical = FALSE
47:  AtomicInc(globalTS)
48:  // stage 3: write-back stage
49:  WRITEBACK(tx)
50:
51: Function WRITEBACK(tx)
52:  while exists an ACTIVE transaction t do
53:    if t.startTS < tx.endTS then
54:      wait
55:  for each copy in tx.log do
56:    memcpy_content(copy.source, copy)
57:    pflush(copy.source.content)
58:    copy.source.next = EMPTY
59:  pfence          // not necessary for correctness
```

three stages. In the *persist stage*, a transaction persists[2] all the object copies in its log into NVM (line 37-38). After that, it retrieves the value of the global timestamp as its log's persist timestamp (line 39) and makes the persistTS persistent (line 40-41). The *pfence* instruction in line 38 guarantees the log's content reaches NVM before its persistTS, and the *pfence* instruction in line 41 ensures both the log and its persistTS reaches NVM. A checksum can be appended to reduce the two fences to one [65]. A transaction's updates become durable once its persistTS reaches NVM (*durable point*). After a crash, a recovery procedure will replay transactions according to the redo logs and in persistTS order.

In the *concurrency commit stage*, the transaction updates its timestamp atomically by updating the 64 bit endTS with the globalTS (line 45). A boolean flag *inCritical* is used to protect this update to make sure the updated endTS is eventually visible to concurrent reads. For example, T_1 may read the globalTS and update its endTS. However, the updated endTS may be kept in the CPU store buffer and waits to be flushed to CPU cache. As a result, a concurrent transaction T_2 whose startTS is not less then T_1's endTS may fail to observe T_1's update. With the *inCritical* flag, T_1 will be blocked until *inCritical* is disabled before it tries to read T_2's endTS (line 14, 15). This ensures T_2's updates on endTS is eventually visible to T_1. As TSO architecture may reorder read-/write instructions, one fence[3] (line 44) is needed to ensure the execution order of line 43 and 45.

In the *write-back stage*, the transaction first waits for all active transactions whose startTS is less than its endTS to finish (line 52-54). This period actually is the *grace period*. It avoids falsely overwriting an original object which may be needed by transactions with small startTS. Because after this period, all threads are either in an inactive state (not executing transactions) or executing transactions with a startTS larger than the original object's timestamp. Therefore, the original object is *dead* which means it is no longer needed by any transactions. At the same time, this period also helps to detect write-write conflicts. Considering another conflicting transaction with smaller startTS, but access the same object after this transaction. This transaction will be blocked at the write-back stage and cannot release the lock. So, the conflicting transaction will abort since it fails to acquire the lock when accessing the object (line 26). At the end of the write-back stage, it writes each next object to the original object (line 56-57) and releases locks by clearing *next* fields (line 58).

Programming: Each transaction should be surrounded by *TM_Start* and *TM_Commit*. For reading/writing an object, it first uses *TM_Read*/*TM_Write* to achieve the ob-

[2] In the current implementation on Intel CPU, Pisces uses *clwb* to flush cacheline and *MFENCE* to ensure previous flushed cachelines reach NVM.

[3] Currently, Pisces uses *MFENCE* [3] instructions which ensures the CPU store buffer is always drained besides serializing load and store operations.

ject pointer and then directly accesses that object with the pointer. Pisces also offers helper functions (*TM_Read_Field* and *TM_Write_Field*) to ease programming.

3.2 Log Recycle

Pisces stores logs in per-thread ring buffers and lets each thread recycle its own logs. Generally speaking, there are two principles for log recycle in Pisces for snapshot isolation and crash consistency, separately.

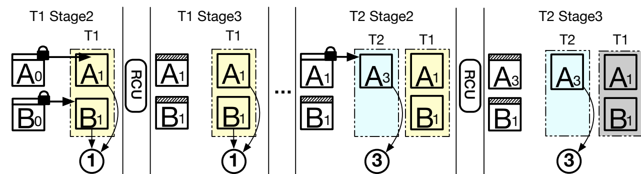


Fig. 4: T1 and T2 are two read-write transactions in one thread. T2 happens after T1. Colored rectangles represent transactions' logs.

P-1: A transaction log can only be recycled after all the copies in it are dead. A transaction creates new versions of objects in its log and exposes them to other transactions. As shown in Figure 4, after a transaction T1 writes the new versions back to the original objects, it is possible that the transaction's log is still required. For example, another transaction can read T1's log if it starts before T1 unlocks the objects and its startTS is no smaller than A's endTS. So, a transaction will not reclaim its own log. Instead, when a following read-write transaction (T2) finishes, the execution thread marks the log of the previous read-write transaction (T1) as reclaimable. Similar to how RCU grace periods can help safely overwriting original objects, the end of the grace period in T2's write-back stage can ensure other transactions no longer access T1's log. Specifically, the end of this grace period ensures (recall line 52-54 in Algorithm 1): previous transactions that may access A1 and B1 in T1's log due to smaller startTS are finished. Therefore, Pisces guarantees all the copies in a log are dead before recycling the log, which achieves *P-1*.

P-2: A transaction log can only be recycled no earlier than all the logs with smaller persist timestamps are recycled. Suppose a transaction A updates an object before another transaction B. If B's log is recycled before A and a crash happens, B's updates will lose after recovery because A will be redone according to its log. To enforce *P-2*, Pisces uses an epoch-based mechanism for recycling logs. It logically distributes logs to epochs according to their persist timestamps. First, an execution thread marks a transaction's log as reclaimable through recording the transaction's persist timestamp as the thread's *reclaim timestamp* (a per-thread variable). Second, an execution thread will atomically advance the global *epoch* when it finds that all the threads' *reclaim timestamp* exceeds the current global *epoch*. Once the global *epoch* increases, the logs belong to the previous epoch are no longer required, and the corresponding log area can be reused.

3.3 Proof Sketch of Snapshot Isolation

A formal proof can be found in [1]. According to the specification of snapshot isolation [8], we prove Pisces is correct by proving the following two theorems are correct.

THEOREM 1 (SNAPSHOT WRITE).

If two transactions update the same object, then one transaction's start TS (short for timestamp) should be greater than another's end TS.

PROOF. Based on the fact that, because of locking (line 26, 58), the conflicting transactions update the same object sequentially, we only need to prove the latter's start TS is always larger than the former's end TS. Pisces ensures this invariant by aborting the latter one when it gets a smaller (illegal) start TS: let's assume both of T_i and T_j access object x and T_i is before T_j. If T_j's start TS is smaller than T_i's end TS, T_i will be blocked by the active T_j at the write back phase (line 52-54). T_j must be active because, by the assumption, it will access x after T_i. Thus, when T_j accesses x, it will find the lock is held by T_i and abort itself. □

Before giving Theorem 2, we first define *the TS of an object* as the end TS of its last writer.

THEOREM 2 (SNAPSHOT READ).

If a transaction T_r reads an object x with timestamp TS_x, then: 1) T_r's start TS is not less than TS_x; and 2) There does not exist a transaction T_w that updates x and its end TS is larger than TS_x, but not greater than T_r's start TS.

PROOF. To prove Pisces holds the first invariant, we show that the x's copy returned by TM_READ must be committed by a transaction whose end TS is not greater than T_r's start TS. First, considering the case TM_READ returns x's original version (line 10, 18). On the one hand, Pisces ensures that, when T_r starts, all objects' original versions have timestamp which is not greater than T_i's start TS. On the other hand, Pisces also forbids any transaction whose end TS is greater than T_i's start TS to overwrite the original version (line 52-54). Next, we consider the case that TM_READ returns x's the next version (line 13, 16). Pisces ensures the invariant by adding an extra constraint that the writer's end TS must be not greater than T_r's start TS (line 15). [4]

Instead of directly showing Pisces holds the second invariant, we prove a variant: if T_w's end TS is not larger than T_r's start TS (assumption), then it is also not larger than TS_x. By the assumption above, we can have T_r starts (line 4) after T_w reads the global TS in commit phase (line 45). Now, let's consider two cases: 1). T_r reads x before T_w unlinks the next version from the object (line 58). Thus T_r is able to get the next version updated by T_w. Because the TSO architecture does not reorder the updates/reads on *endTS* and *inCritical* in TM_COMMIT/TM_READ, thus if T_r finds T_w's *inCritical* is false then it must be able to observe T_w's *endTS*. As a

[4]The detail proof of the consistency between the *next* copy and its *writer* pointer can be found in [1].

result, T_w's end TS should be equal to TS_x 2). T_r reads x after T_w writes back and unlinks the next copy. For this case, T_r will read the version committed by T_w or the transaction accessing x after TS_w. Then, we can have TS_x that must be less than T_w's end TS by simply deriving from Theorem 1. □

3.4 Crash Consistency

A recovery procedure will start after a crash and redo the durable transactions in a non-decreasing order of their logs' persistTS. The recovery time is affected by the length of the log which is decided by the log recycling frequency. By default, each execution thread in Pisces tries to recycle the logs after executing 3 (the default threshold) read-write transactions, so Pisces does not suffer from a high recovery cost. Also, Pisces provides a persistent allocator based on SSMalloc [50], which can recover the allocation information.

The key to ensuring crash consistency is that *dependent* transactions are (i) persisted and (ii) redone (after a crash) in the correct order that corresponds to their commit order. Two transactions are *dependent* if the read set or write set of the subsequent one overlaps with the write set of the previous one. In this subsection, we explain how Pisces achieves both (i) and (ii).

Achieving (i): Pisces guarantees the persistence ordering of dependent transactions by deferring the visibility of the updates of a transaction. Specifically, a transaction reaches its durable point (the end of *persist stage*) before it is visible (the end of *concurrency commit stage*). If transaction B observes (depends on) transaction A' updates (visible), A must already be persisted (durable). So, the persistence ordering of two dependent transactions must correspond to their commit order, as shown in Figure 5.

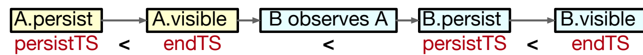


Fig. 5: Transaction B depends on A. Arrows mean happen-before.

Achieving (ii): On one hand, Pisces guarantees that if a transaction B depends on another transaction A, B's persist timestamp must be greater than A's (see Figure 5). On the other hand, Pisces's recovery procedure redoes the transactions' logs in a non-decreasing order of their persist timestamps after a crash. Therefore, transaction B that depends on A will be redone after A, which also corresponds to their commit order (B's endTS is greater than A's.) Besides, independent transactions that have the same persist timestamp can be redone in any order.

4 Optimizations

Flush-diff. Creating redo logs in the granularity of whole objects can avoid searching for new values in the address-value pairs. But it is quite expensive when the modification to an object is much smaller than the object size because the whole log (entire copy) has to be persisted into NVM. Therefore, we give an optimization named *flush-diff* that only persists the modifications. A transaction creates the object copies in DRAM instead of NVM and only records the updates (address-value pairs) in NVM. In the first commit stage, a transaction only needs to flush those updates into NVM; In the second commit stage, it lets the in-DRAM object copies (new versions) become readable; In the third commit stage, it only needs to apply the logged updates to the original objects in the NVM. Therefore, *flush-diff* can still embrace the advantage of logging a whole object (i.e., directly read/write the copy without the overhead of indirection) by keeping the volatile object copy in DRAM, and significantly reduce the amount of NVM persistence operations.

It is worth mentioning that DRAM footprint in Pisces is limited and not related to the amount of NVM in use. This is because Pisces only temporarily buffers the new object versions in DRAM and timely recycles the transactions' logs. Therefore, the DRAM needed for *flush-diff* is only related to the working set size of the transactions whose newly created copies may still be referenced.

Group Commit. Pisces chooses to eagerly write a new version back to an object's home location, which is for benefiting readers but makes the last commit stage for write transactions heavier. As mentioned in Section 3.1, a read-write transaction uses the RCU-like waiting mechanism to overwrite the objects in the home location. Besides, it is also costly to update *globalTS* with atomic instruction like *fetch_and_add* in a high-contention workload.

To amortize the overhead of both RCU reclamation and updating *globalTS*, Pisces batches and commits several write transactions together. An execution thread delays the commitment of a write transaction till the number of pending transactions (wait for commit) reaches a threshold or another transaction needs to update the same object with T. Then, the thread commits the pending transactions together, i.e., execute RCU mechanism and update the globalTS for one time. Thus, the overhead is amortized by these transactions.

5 Evaluation

5.1 Experimental Setup

Basic Setup. We conduct the experiments on a server provided by Intel. The server has two sockets, each containing a 10-core Intel Xeon Gold 5215M CPU, 128GB DRAM and 128GB Intel Optane DC Persistent Memory (NVM). We enable hyper-threading and bind each software thread to each hyper-thread (40 hyper-threads in all) that runs at 2.5GHz. The Linux kernel version is 4.19.32, and the GCC version is 8.3.1. Without an explicit statement, we use *clwb* instructions to persist data into NVM.

System for Comparison. *DUDETM* [49] maps persistent data in NVM to DRAM and uses TinySTM [35] which is a word-based STM to execute transactions in DRAM. A transaction only needs to write logs in a per-thread volatile log

buffer, and the background persist threads flush these volatile logs into persistent log buffer. DUDETM also requires another reproduce thread to write logs back to the persistent data. When evaluating DUDETM, we set the size of pre-allocated DRAM area the same as pre-allocated NVM area to make it able to cache all the NVM data in DRAM, which avoids the potential high overhead of page swapping. The volatile log buffer for each thread can hold 1 million log entries (default configuration), and we double the default size of persistent log buffer to make it able to hold 32 million log entries. In all the experiments for DUDETM, we wait for only foreground threads and not for background threads to finish. Thus, the committed transaction may be not durable yet. Besides, the background threads are not counted into the thread number. Currently, DUDETM does not implement multiple background persistent threads, but the support can be added without changing its design. However, the background reproduce thread needs to replay transactions' logs according to the unique transaction ID, i.e., a total order (hard to utilize multi-threading), which will become the bottleneck in some scenarios. To avoid the single persist thread becoming the bottleneck in the experiments, we allocate the persistent log buffer (should be in NVM actually) in DRAM.

Benchmarks. Generally speaking, to develop SI-safe applications, programmers need to analyze whether write skew anomalies may happen in some cases and then avoid the potential anomalies through making write-write conflicts in such cases. We manually ensure the presented benchmarks are SI-safe. referring to [48, 53, 54]. For examples, a transaction on a linked list (used in the following hash tables) adds all the modified nodes *including the nodes to be removed* into its write set in which each node object will be locked before updated; a transaction on the following tree structures traverses from the root node to some node in one way and also adds all the to-be-modified nodes into its write set.

5.2 Micro-benchmarks

Hash Table. Each hash table contains 10K buckets (implemented as linked lists) and initially contains 100K key-value pairs. We create different numbers of threads to execute search and insert/remove transactions. Figure 6 presents the evaluation results with various update rates. Note that the y-axis uses *log-scale*. Every test runs 30 seconds.

When the update rate is low, such as 0% and 2%, both Pisces and DUDETM scale well. Although Pisces's read-only transactions may load data from NVM when the data does not reside in CPU cache, they are still faster than that in DUDETM. It is because DUDETM requires a software page table mechanism for translating NVM addresses to DRAM addresses and validation for read operations. Different from DUDETM which incurs software overhead for read operations, Pisces embraces a read-friendly design, and thus its throughput is much higher and grows faster.

When the update rate is 20%, Pisces still scales well within

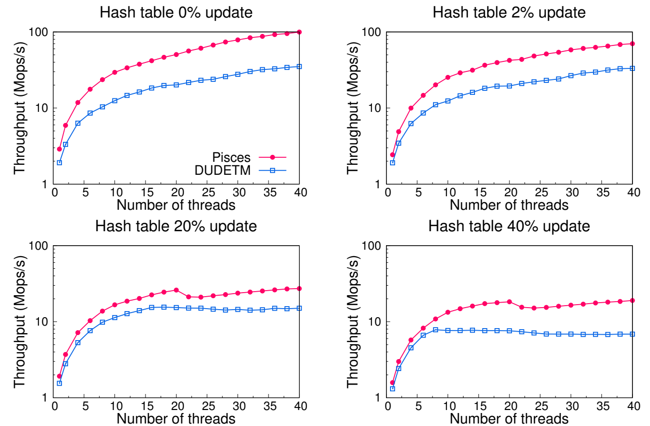


Fig. 6: The throughput of hash table (8-byte key and 64-byte value) at various update rates (legends in the first figure).

20 threads, i.e., a single NUMA (non-uniform memory access) node. The throughput of Pisces increases from ∼1.9 Mops/s to ∼26 Mops/s when the thread number increases from 1 to 20. However, there is an obvious performance drop when the thread number changes from 20 to 22. The main cause is cross-NUMA memory accesses. First, a global timestamp is updated in read-write transactions in Pisces, and a large number of read-write transactions incur high contention on that timestamp. Second, the grace period detection is another overhead for the read-write transactions, and its cost can be enlarged by frequent remote memory accesses. Recall that Pisces uses the RCU/RLU grace period detection mechanism (line 52-54 in Algorithm 1) to avoid overwriting in-use objects' versions. Therefore, while the total throughput of Pisces still grows as the thread number increases from 22 to 40, the growth speed is much slower than that within a single NUMA node.

The throughput of DUDETM grows from ∼1.5 Mops/s to ∼15 Mops/s as the thread number increases from 1 to 16. However, DUDETM's throughput cannot keep growing or even decreases a little when the thread number becomes larger. The reason is that the background threads fails to timely clean up the logs of transactions. Note that the reproduce thread has to modify the persistent objects according to the logs in the order of transaction execution, and make the modifications persistent with cacheline flush instructions. In contrast, Pisces lets each execution thread to persist and write-back the transactions' logs, which can better utilize the NVM write bandwidth.

Table 1: The average cost of a read-write transaction and one grace period detection in Pisces's hash table. The update rate is 40%.

#Thread	10	20	30	40
RW TX Latency (cycles)	2902	3986	7180	8496
Grace Period (cycles)	357	790	1689	2140

The performance of DUDETM and Pisces at 40% update rate shows similar trends with those at 20% update rate. Nev-

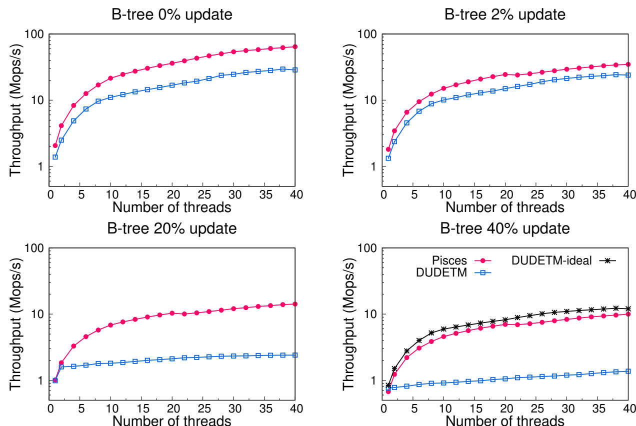


Fig. 7: The throughput of B+-tree whose node size is 256 bytes with various update rates (legends in the last figure).

ertheless, the throughput of DUDETM only grows within 8 threads because a higher update rate means more read-write transactions (generate more logs and fill the log buffer earlier). For Pisces, the growth speed of throughput becomes lower when the thread number exceeds 20. As presented in Table 1, the cost of grace period increases as the thread number increases. The reason is one thread has to check other threads' status for detecting the grace period.

In the case of 40 threads, Pisces's throughput is about 1.8× and 2.7× of DUDETM's when the update rate is 20% and 40%, respectively. And, its persistency cost is 19% at 40% update rate. Besides, the abort rates of both Pisces and DUDETM (if no blocking) are nearly zero since the hashing mitigates the contention among different threads. However, DUDETM's abort rate increases (up to 9%) if blocking happens because a thread may get blocked with holding locks.

We also evaluate the hash table with an occupancy of 0.75, i.e., 10K buckets and 7.5K key-value pairs. The evaluation results show similar trends. Specifically, at 20% update ratio, the throughput of DUDETM grows from ∼3.3 Mops/s to ∼15.9 Mops/s as the thread number increases from 1 to 16. As before, its throughput cannot grow when the thread number is larger than 16. Pisces's throughput grows from ∼2.7 Mops/s to ∼35.9 Mops/s as the thread number increases from 1 to 20. Nevertheless, when the thread number is 1 or 2, DUDETM has a higher throughput than Pisces for two reasons: first, DUDETM leverages extra CPUs (background threads) for persisting data; second, each transaction reads fewer data due to lower occupancy, which mitigates the benefits of read-friendly design in Pisces.

B+-tree. We construct B+-trees in which each node contains at most 16 children and randomly insert about 1 million key-value pairs at the beginning of each test. Figure 7 shows the evaluation results of executing search and insert transactions (an insert transaction will modify the target key-value pair if the pair already exists) on B+-trees. Each transaction goes down from the root node to some leaf node. For an insert transaction, before going down to some node, it first checks if the node is full. If the node is full, it splits the node for creating space. Since we only implement delete operations as marking the target node as deleted, we run every test for 10 seconds in case the trees get too large.

Similar to the results of hash table benchmark, both Pisces and DUDETM can scale well to 40 threads when the update rate is low such as 0% and 2%, because the number of NVM writes is small. Owing to the read-friendly designs, Pisces shows a better performance than DUDETM. Nevertheless, the performance gap between Pisces and DUDETM decreases when the update rate changes from 0% to 2% because Pisces synchronously persists data into NVM while DUDETM hides the persistence overhead through asynchronously persisting the data in the background.

At 20% update rate, DUDETM's throughput is almost the same as Pisces's when there is a single execution thread. However, DUDETM can only scale to two threads while Pisces has much better scalability. The scalability issue of DUDETM arises earlier in B+-tree benchmark than in hash table benchmark because the read-write transactions in B+-tree generates more log and thus burden the reproduce thread more. Enlarging the size of log buffer can mitigate/hide the problem to some extent but cannot eliminate/solve this problem. At 40% update rate, DUDETM actually outperforms Pisces when the thread number is 1. However, its scalability issue gets worse because of the higher update rate.

At 40% update rate, DUDTEM's throughput grows from 763 Kops to 1370 Kops as the thread number increases from 1 to 40. The reason is each execution thread has one volatile log buffer. Therefore, the total throughput grows a little when adding more threads (more buffer). Nevertheless, longer running time will further flatten the throughput.

To clearly show that the reproduce thread blocks the execution threads and restricts the overall performance, we also evaluate the performance of *DUDETM-ideal* in which the background reproduce thread directly marks the persistent log area as free without writing back the logs in it to persistent objects in NVM. In fact, *DUDETM-ideal* emulates the performance of DUDETM with an infinite persistent log buffer. As shown in the last sub-figure in Figure 7, *DUDETM-ideal* scales very well to 38 threads. The reason for the performance drop in 40 threads is the total thread number (40 execution threads together with 2 background threads) exceeds the hardware thread number (40 hyper-threads). Since the only difference between *DUDETM-ideal* with DUDETM is whether the background reproduce thread really flushes data to NVM, we can conclude the centralized reproduce thread severely harms the system's scalability.

When the update rate is 20% or 40%, the throughput of Pisces almost keeps growing as the thread number increases to 40. And, the persistency cost is 36% at 40% update rate. The NUMA problem in B+-tree is less severe than that in the hash table since the total throughput is lower in B+-tree

Table 2: The average cost of a read-write transaction and one grace period detection in Pisces's B+-tree. The update rate is 40%.

#Thread	10	20	30	40
RW TX Latency (cycles)	9,374	12,024	15,210	16847
Grace Period (cycles)	1,162	2,021	2,915	3528

(less read-write transactions). Nevertheless, the performance growth speed still becomes slower when the thread number exceeds 20. Table 2 presents the cost of grace period detection in Pisces's B+-tree.

5.3 Real-world Benchmarks

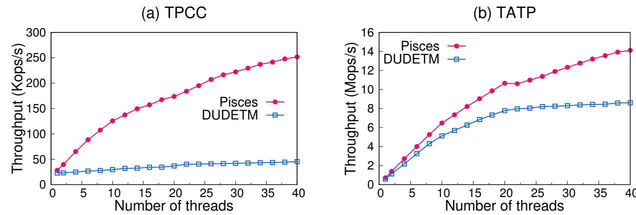


Fig. 8: (a) TPC-C new-order transactions and (b) TATP.

We also evaluate macro-benchmarks (i.e., TPC-C and TATP) which are tested in DUDETM. Besides, we further evaluate Pisces and DUDETM on *kmeans*, *ssca2*, and *vacation* which are popular transactional memory benchmarks [57, 58].

TPC-C. TPC-C is an online transaction processing (OLTP) benchmark. We implement its *new-order transaction* [24] with B+trees whose nodes contain at most 32 children as the tables. In this experiment, each execution thread works on its corresponding warehouse and executes new-order transactions (the update rate is 100%). On average, each transaction involves inserting over 10 new objects into different tables as well as modifying over 10 existing objects, which generates much more logs than the transactions in the previous micro-benchmarks.

Figure 8-(a) gives the evaluation results. Since there is no conflict among transactions from different execution threads, the throughput of Pisces continues to grow with the increase of thread number. There are no transaction aborts in this experiment for Pisces. For fairness, we also modify the TinySTM used by DUDETM (enlarge the *LOCK_ARRAY_LOG_SIZE*) to avoid false sharing of address locks and reduce the abort rate in DUDETM to zero. However, the throughput of DUDETM at 40 threads is only twice of that at 1 thread.

The evaluation results clearly demonstrate that Pisces utilizes the NVM bandwidth in a much better way than DUDETM. A centralized log-reproducing thread can hardly catch up with the progress of multiple execution threads. So the execution threads fill the log buffers in DUDETM. Once the log buffers are full, all the execution threads are blocked, and then the whole system's progress relies on the background log-reproducing thread. Instead of flushing data to NVM in a centralized way, Pisces lets each thread make the transactions' persistent and thus allows more parallelism in the NVM persistence operations. Since NVM device can serve the memory operation requests from different CPUs at the same time, the throughput of Pisces grows from ∼28 Kops/s to ∼252 Kops/s as the thread number increases from 1 to 40. The NVM hardware bandwidth limit is still not reached, inferred from the growth in throughput.

Compared with DUDETM, Pisces can achieve about 4.6× speedup when the thread number is 40. And, the persistency cost is about 50%.

TATP. TATP benchmark [72] is another OLTP application. We implement three read-only transactions and three read-write transactions of it. We use the same B+Tree in the TPC-C experiments as the data structure of tables, set the update ratio to 18% and initialize the population size to 200,000.

Different from the new-order transactions in TPC-C, the read-write transactions in TATP are much smaller. For example, the *update-location transaction* that occupies 14% of the total transactions only update one single existing object. So, each TATP read-write transaction involves less NVM writes than TPC-C transactions and even less than the B+-tree micro-benchmark in which most read-write transactions insert a new object. Thus, DUDETM can scale to 22 threads. However, its scalability issue still comes up when the thread number gets larger.

For Pisces, the throughput grows from ∼0.7 Mops/s to ∼14 Mops/s as the thread number increases from 1 to 40. Pisces's peak performance is about 64% higher than DUDETM's. And, the persistency cost is 26%. The NUMA issue also appears in this benchmark. The RCU grace period detection cost is higher if the total throughput of read-write transactions is higher because of the higher (cross NUMA nodes) cacheline contention (i.e., checking other thread's status). This is also why the NUMA issue is not obvious in the TPC-C benchmark. The NUMA issue also leads to a slower growth speed of the throughput after the thread number exceeds 20.

TM Applications. Referring to [7], we implement and evaluate *kmeans*, *ssca2*, and *vacation* in both Pisces and DUDETM. Currently, we only persist data that are surrounded/protected by TM interfaces.

Table 3: The execution time of *kmeans* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	5.5	5.0	3.8	2.6	0.25	0.54
Pisces (s)	3.2	2.4	1.3	0.8	0.7	1.2

Kmeans is a machine learning application and this experiment test it with low contention and medium data set. Table 3 shows the execution time. In this benchmark, both DUDETM and Pisces scale to 16 threads. However, there is a performance drop for both systems when the thread number

increases from 16 threads to 32 threads, because they both suffer from high abort rates (84% for DUDETM and 77% for Pisces). The performance of Pisces is also bottlenecked by grace periods' cost which is enlarged by NUMA. It is also worth mentioning that the foreground threads in DUDETM get blocked by full volatile logs (slow background threads) when the thread number is less than 8. As a result, DUDETM has an obvious performance improve when the thread number increases from 8 to 16. For the same reason, DUDETM performs worse than Pisces with no more than 8 threads. When there are more than 8 threads, DUDETM has better performance than Pisces because the foreground threads in DUDETM neither need to persist write transactions nor get blocked by background threads.

In this benchmark, with the increase of thread number, foreground threads in DUDETM are less likely to get blocked by the reproduce thread. The reason is that *kmeans* evenly distribute a specific amount of work to the foreground execution threads and thus each thread executes less transactions when there are more threads. Specifically, to finish this benchmark, all the threads need to commit 1M transactions in total and generate 40M log entries. When the thread number is more than 16, the foreground threads in DUDETM do not get blocked since each of them has a volatile buffer with 1M log entries. Nevertheless, when testing with large data set (10M transactions and 400M log entries), the foreground threads in DUDETM still get blocked when there are 32 threads.

Table 4: The execution time of *ssca2* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	17.2	13.1	12.0	11.6	9.3	9.3
Pisces (s)	18.8	13.8	8.4	5.4	3.6	3.7

Scalable Synthetic Compact Applications (*ssca2*) simulates the computation on graphs. Table 4 gives the evaluation result of ssca2 with medium data set (2^{18} nodes). Different from kmeans, ssca2 involves a larger number of transactions and DUDETM cannot scale well since the background reproduce thread cannot timely consume the logs. So for DUDETM, 32 threads cannot finish this benchmark faster than 16 threads since the execution threads get blocked. Nevertheless, if evaluating ssca2 benchmark with the small data set (2^{13} nodes), DUDETM can scale well but the performance of Pisces also gets much better.

Pisces scales better than DUDETM in this benchmark. However, the execution time of Pisces is longer than that of DUDETM when the thread number is 1 and 2. This is because foreground threads in DUDETM do not make the transactions' updates persistent, and we only calculate the runtime of the foreground threads. Similar to previous experiments, Pisces suffers from NUMA problem again. Since the throughput in this benchmark is high (executes ∼11M transactions in total) and the update rate is 100%, the NUMA problem is more severe and thus causes the performance to drop when threads number changes from 16 to 32.

Table 5: The execution time of *vacation* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	10.0	5.2	2.5	1.2	0.8	0.4
Pisces (s)	8.2	4.7	2.7	1.6	0.9	0.6

Vacation is an OLTP system which emulates a travel reservation system. The *vacation* benchmark has 100% update ratio, and each transaction has bigger read/write sets. We use hash tables to implement tables in the benchmark. Table 5 shows the evaluation result of this benchmark with medium data set and low contention.

As shown in Table 5, the execution time of both DUDETM and Pisces decreases as the thread number increases in this benchmark. In the case of 1 and 2 threads, Pisces performs better than DUDETM. The reason is that DUDETM introduces software overhead for the read operations in the transactions. Since the read sets are large, the software overhead such as read validation is non-negligible. But DUDETM scales well since this workload (executes 400K transactions in total) does not cause the full log problem.

Overall, Pisces does not scale as well as DUDETM. As the benchmark is update-only and the average transaction latency is long (big transaction), the grace period detection mechanism in Pisces becomes more time-consuming and brings high overhead. Hence, with the increase of thread number, transactions spend more time in the grace period detection mechanism, which leads to dissatisfactory performance. On average, the mechanism costs each transaction 30,433 cycles and 46,070 cycles when the thread number is 16 and 32, respectively. And the average transaction latency is about 168,000 cycles when there are 32 threads.

5.4 Other Performance Analysis

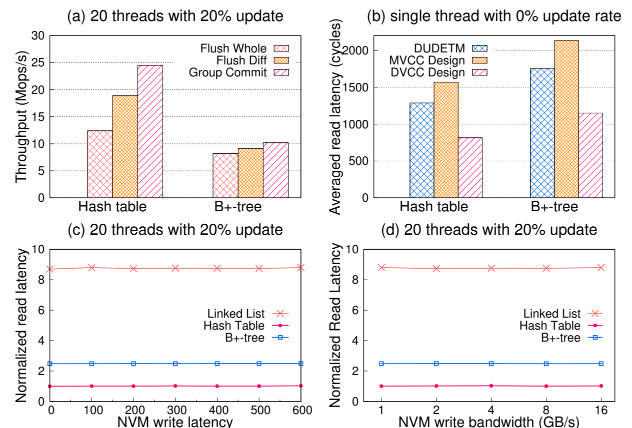


Fig. 9: (a) The performance gain break down. (b) The performance of read-only transactions. Read latency in Pisces with (c) increased NVM write latency and (d) increased NVM write bandwidth.

Figure 9-(a) shows step-wise performance gain from the

optimizations in Pisces. *Flush Diff* improves the throughput of hash table and B+-tree by 52% and by 11%, respectively, because it effectively reduces the number of NVM writes. Since a write transaction on a hash table changes a small portion of the nodes, there is more data that does not need to be logged and written back. However, a transaction on a B+-tree usually involves more data modification, especially, when node splitting is required. So *Flush Diff* benefits hash table more in the presented settings. *Group Commit* (setting group number as 3) brings a performance gain of 30% for hash table and 12% for B+-tree, through reducing the number of grace period detection and *fetch_and_add* instructions on the global timestamp. The performance improvement in hash table is larger for two reasons. First, Pisces has a higher throughput in the hash table benchmark which means higher contention on the global timestamp. Second, the grace period detection in hash table takes a higher percentage of cost in a write transaction than that in B+-tree. Note that the current *Group Commit* implementation will increase the latency of write transactions. Nevertheless, Pisces can only batch the write-back stages instead of the whole commit phases to mitigate this issue.

Figure 9-(b) compares the average latency of read-only transactions in micro-benchmarks. Compared with our MVCC-based design, the DVCC design in Pisces significantly reduces the read latency (about 2× faster). The reason is that each read operation in the MVCC-based design involves locating the version list of an object and traversing the list, leading to at least one more random memory access (i.e., read indirection). Compared with DUDETM (needs address translation and read validation), Pisces's read operations are also faster owing to the read-friendly design. For the hash table benchmark, the read-only transaction in Pisces is faster than DUDETM's by 472 cycles. While for the B+-tree benchmark, the read-only transaction in Pisces is faster than DUDETM's by 605 cycles. This is because a read transaction in B+-tree contains more read operations.

Different persistent memory technologies may have different persistent cost. We further use an NVM emulator which explicitly add delays to the NVM flush operations according to the NVM write latency and bandwidth (similar to prior work [17, 37, 49, 49]). As shown in Figure 9-(c) and Figure 9-(d), the average latency[5] of read-only transactions, each of which searches for an element in the corresponding data structure, is stable with various NVM write latency and bandwidth. The reason is that Pisces avoids writers blocking readers. Although the NVM write latency and bandwidth affect the latency of write operations in Pisces, the latency of read-only transactions is insensitive to that of write operations. Therefore, Pisces produces a stable average latency of read-only transactions with increased NVM write latency and bandwidth. Other experiments with different thread numbers or different update rates give similar results.

[5]We set the read latency in hash table in the case of zero NVM write latency to 1. Other results are normalized against it.

6 Related Work

Compared with most PTM designs (Mnemosyne [74], NV-Heaps [21], Kamino-Tx [56], and DCT [44]) which reduce the persistence latency of write transactions through various novel techniques but may expose NVM persist overhead to readers, Pisces focuses on benefiting read operations and can always hide NVM persist latency from readers. A most recent PTM named Romulus [23] promises never blocking read-only transactions through maintaining twin copies of the durable data. Pisces shares a similar idea to avoid blocking read-only transactions and further exploits SI to avoid blocking any read operation. Romulus instruments loads and stores to NVM through programming language feature, which is elegant and can be borrowed to Pisces. Besides, Romulus chooses a single writer design which can reduce the average number of fences for write transactions, however, limits the concurrency of NVM persistence. Some recent studies [42, 60, 71] leverage hardware modifications to implement efficient PTM systems. NVM is also exploited by in-memory database systems [27, 43] and new file systems [31, 32, 80]. Others [5, 37, 40, 41, 47] provide libraries for applications to utilize NVM.

Transactional Memory (TM) has been well studied [10, 14, 29, 34, 39, 66, 68, 70]. Some studies [36, 73] propose non-blocking designs and some others also investigate snapshot isolation to TM [48, 68], which significantly reduces the abort rates. But they do not consider crash consistency under NVM. Matveev et al. [54] propose a novel and lightweight synchronization mechanism (RLU) for concurrent programming. DVCC in Pisces is inspired by both MVCC and RLU. Thus, Pisces and RLU share a couple of similarities including maintaining two versions and allowing readers to read the write sets of writers. However, RLU neither provides a transactional programming semantic (no snapshot isolation) nor considers NVM (no durability and crash consistency).

7 Summary

This paper presents Pisces, a read-friendly PTM that provides transactional memory APIs for programming on NVM. With several techniques such as DVCC and three-stage commit, Pisces achieves both high throughput and good scalability while ensuring snapshot isolation and crash consistency.

8 Acknowledgement

We sincerely thank our shepherd Michael Swift and the anonymous reviewers for their insightful suggestions. Also, we sincerely thank the authors of DUDETM for giving guidance. This work is supported in part by China National Natural Science Foundation (No. 61672345). Zhaoguo Wang is the corresponding author.

References

[1] The formal proof on the snapshot isolation guarantee. https://ipads.se.sjtu.edu.cn/_media/publications/guatc19_proof.pdf.

[2] Intel 3dxpoint technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[3] Intel 64 and ia-32 architectures software developer's manual. https://software.intel.com.

[4] Micron 3dxpoint technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[5] Persistent memory development kit, nvml. http://pmem.io/pmdk/.

[6] Redis. https://redis.io/.

[7] The stanford transactional applications for multi-processing; a benchmark suite for transactional memory research. https://github.com/kozyraki/stamp.

[8] Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.

[9] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[10] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 4–18. Springer, 2009.

[11] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342. ACM, 2015.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[13] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[14] Jayaram Bobba, Neelam Goyal, Mark D Hill, Michael M Swift, and David A Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 127–138. IEEE Computer Society, 2008.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[16] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009.

[17] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[18] Haibo Chen, Heng Zhang, Ran Liu, Binyu Zang, and Haibing Guan. Fast consensus using bounded staleness for scalable read-mostly synchronization. *IEEE Transactions on Parallel & Distributed Systems*, (12):3485–3500, 2016.

[19] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[20] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[21] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[22] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[23] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.

[24] Transaction Processing Performance Council. http://www.tpc.org/tpcc/. *TPC Benchmark C*.

[25] Transaction Processing Performance Council. http://www.tpc.org/tpce. *TPC Benchmark E*.

[26] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726. VLDB Endowment, 2006.

[27] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael

Stonebraker, Stan Zdonik, and Subramanya Dulloor. A prolegomenon on oltp database systems for non-volatile memory. *ADMS@ VLDB*, 2014.

[28] Cristian Diaconu, 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*, pages 1243–1254. ACM, 2013.

[29] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[30] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[31] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 719–731. USENIX Association, 2017.

[32] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[33] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[34] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[35] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[36] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[37] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[38] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[39] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 31. ACM, 2016.

[40] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[41] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC'17). Santa Clara, CA. http://jinglei. ren. systems/files/lsnvmm_slides_atc17. pptx*, 2017.

[42] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the International Symposium on Computer Architecture*, 2018.

[43] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[44] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

[45] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE, 2013.

[46] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1), 2010.

[47] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Jónsson, and Laurent Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):869–883, 2009.

[48] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. Si-tm: reducing transactional memory abort rates through snapshot isolation. *ACM SIGARCH Computer Architecture News*, 42(1):383–398, 2014.

[49] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–343. ACM, 2017.

[50] Ran Liu and Haibo Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 15. ACM, 2012.

[51] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *USENIX Annual Technical Conference*, pages 219–230, 2014.

[52] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[53] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[54] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2015.

[55] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[56] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *EuroSys*, pages 499–512, 2017.

[57] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

[58] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 135–148. ACM, 2017.

[59] 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*, pages 677–689. ACM, 2015.

[60] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 336–349. IEEE, 2018.

[61] Lois Orosa and Rodolfo Azevedo. Logsi-htm: Log based snapshot isolation in hardware transactional memory.

[62] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 265–276. IEEE Press, 2014.

[63] Hasso Plattner. The impact of columnar in-memory databases on enterprise systems: implications of eliminating transaction-maintained aggregates. *Proceedings of the VLDB Endowment*, 7(13):1722–1729, 2014.

[64] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[65] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP ’05, pages 206–220, New York, NY, USA, 2005. ACM.

[66] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[67] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[68] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT’06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

[69] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*, pages 245–258. ACM, 2018.

[70] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[71] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: a flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190. ACM, 2017.

[72] Neuvonen Simo, Wolski Antoni, manner Markku, and Raatikka Vilho. http://tatpbenchmark.sourceforge.net/. *Telecom Application Transaction Processing Benchmark*.

[73] Fuad Tabba, Mark Moir, James R Goodman, Andrew W Hay, and Cong Wang. Nztm: Nonblocking zero-indirection transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 204–213. ACM, 2009.

[74] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[75] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, 2017.

[76] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1):58–61, 2015.

[77] Wikipedia. https://en.wikipedia.org/wiki/Snapshot_isolation. *Snapshot isolation*, 2017.

[78] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[79] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 349–362. USENIX Association, 2017.

[80] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[81] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

EdgeWise: A Better Stream Processing Engine for the Edge

Xinwei Fu, *Virginia Tech*
Talha Ghaffar, *Virginia Tech*
James C. Davis, *Virginia Tech*
Dongyoon Lee, *Virginia Tech*

Abstract

Many Internet of Things (IoT) applications would benefit if streams of data could be analyzed rapidly at the Edge, near the data source. However, existing Stream Processing Engines (SPEs) are unsuited for the Edge because their designs assume Cloud-class resources and relatively generous throughput and latency constraints.

This paper presents EdgeWise, a new Edge-friendly SPE, and shows analytically and empirically that EdgeWise improves both throughput and latency. The key idea of EdgeWise is to incorporate a congestion-aware scheduler and a fixed-size worker pool into an SPE. Though this idea has been explored in the past, we are the first to apply it to modern SPEs and we provide a new queue-theoretic analysis to support it. In our single-node and distributed experiments we compare EdgeWise to the state-of-the-art Storm system. We report up to a 3x improvement in throughput while keeping latency low.

1 Introduction

Internet of Things (IoT) applications are growing rapidly in a wide range of domains, including smart cities, healthcare, and manufacturing [33, 43]. Broadly speaking, IoT systems consist of Things, Gateways, and the Cloud. Things are sensors that "read" from the world and actuators that "write" to it, and Gateways orchestrate Things and bridge them with the Cloud.

At the moment, IoT systems rely on the Cloud to process sensor data and trigger actuators. In principle, however, Things and Gateways could perform some or all data analysis themselves, moving the frontier of computation and services from the network core, the Cloud [17], to its Edge [21, 67], where the Things and Gateways reside. In this paper we explore the implications of this paradigm in a promising use case: *stream processing*.

Stream processing is well suited to the IoT Edge computing setting. Things generate continuous streams of data that often must be processed in a timely fashion; stream processing performs analysis on individual data points (*tuples*) rather than batches [24, 69]. As shown in Figure 1, stream processing is described by a directed acyclic graph, called a *topology*, whose vertices are data processing *operations* and edges indicate data flow.

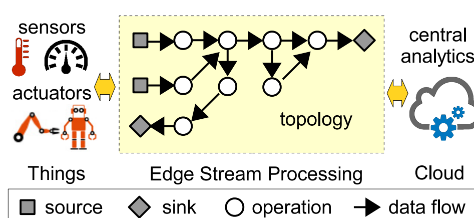


Figure 1: The Edge connects Things to the Cloud, and can perform local data stream processing.

Modern Stream Processing Engines (SPEs) such as Storm [15], Flink [13], and Heron [49] have mostly been designed for the Cloud, assuming powerful computing resources and plenty of memory. However, these assumptions do not hold at the Edge. In particular, these SPEs use a simple One Worker Per Operation Architecture (OWPOA). Given a dataflow topology and the degree of parallelism of each operation, OWPOA-style SPEs assign a dedicated *worker* thread to each *operation* instance, and link the worker-operation pairs with queues. Then, they rely on the operating system (OS) scheduler to choose which worker (operation) to schedule next, leading to *lost scheduling opportunities*: data propagates haphazardly through the topology because the scheduling of worker threads is left to the congestion-oblivious OS. With Cloud-scale resources these inefficiencies are amortized, but at the Edge they cannot be.

The database community studied efficient operation scheduling about a decade before modern SPEs came into vogue [18, 25]. Sadly, however, lessons learned from this

early research were not carried into the design of modern SPEs, leading to a sub-optimal performance when the existing OWPOA-style SPEs are used at the Edge setting. Existing IoT computing literature (e.g. [58, 64]) has ignored this history and has been building Edge computing platforms based on unmodified modern SPEs. This paper explores the impact of applying to modern SPEs these "lost lessons" of operation scheduling.

We present EdgeWise, an Edge-friendly SPE that revives the notion of engine-level operation scheduling to optimize data flows in a multiplexed (more operations than processor cores) and memory-constrained Edge environment. EdgeWise re-architects SPE runtime design and introduces an engine-level scheduler with a fixed-size worker pool where existing *profiling-guided* scheduling algorithms [18, 25] may be used. EdgeWise also proposes a *queue-length-based* congestion-aware scheduler that does not require profiling yet achieves equivalent (or better) performance improvement. EdgeWise monitors the numbers of pending data in queues, and its *scheduler* determines the highest-priority operation to process next, optimizing the flow of data through the topology. In addition, the EdgeWise's worker pool avoids unnecessary threading overheads and decouples the data plane (operations) from the control plane (workers).

We show analytically and empirically that EdgeWise outperforms Apache Storm [15], the exemplar of modern SPEs, on both throughput and latency. EdgeWise is a reminder of both the end-to-end design principle [65] and the benefits of applying old lessons in new contexts [61].

This paper provides the following contributions:

- We study the software architecture of existing SPEs and discuss their limitations in the Edge setting (§3). To the best of our knowledge, this paper is the first to observe the lack of operation scheduling in modern SPEs, a forgotten lesson from old SPE literature.
- We present EdgeWise, a new Edge-friendly SPE. EdgeWise learns from past lessons to apply an engine-level scheduler, choosing operations to optimize data flows and leveraging a fixed-size worker pool to minimize thread contention (§4).
- Using queuing theory, we argue analytically that our congestion-aware scheduler will improve both throughput and latency (§5). To our knowledge, we are the first to mathematically show balancing the queue sizes lead to improved performance in stream processing.
- We demonstrate EdgeWise's throughput and latency gains on IoT stream benchmarks (§7).

2 Background: Stream Processing

This section provides background on the stream processing programming model (§2.1) and two software architectures used in existing SPEs (§2.2).

2.1 Dataflow Programming Model

Stream processing uses the dataflow programming model depicted in the center of Figure 1 [24, 69]. Data *tuples* flow through a directed acyclic graph (*topology*) from *sources* to *sinks*. Each inner node is an *operation* that performs arbitrary computation on the data, ranging from simple filtering to complex operations like ML-based classification algorithms. In the Edge context a source might be an IoT sensor, while a sink might be an IoT actuator or a message queue to a Cloud service.

Though an operation can be arbitrary, the preferred idiom is *outer I/O, inner compute*. In other words, I/O should be handled by source and sink nodes, and the inner operation nodes should perform only memory and CPU-intensive operations [3, 41]. This idiom is based on the premise that the more unpredictable costs of I/O will complicate the scheduling and balancing of operations.

After defining the topology and the operations, data engineers convert the *logical* topology into a *physical* topology that describes the number of physical instances of each logical operation. In a distributed setting engineers can also indicate preferred mappings from operations to specific compute nodes. An SPE then deploys the operations onto the compute node(s), instantiates queues and workers, and manages the flow of tuples from one operation to another.

2.2 Stream Processing Engines

Starting from the notion of active databases [55, 76], early-generation SPEs were proposed and designed by the database community in the early 2000s: e.g. Aurora [24], TelegraphCQ [27], Stream [16], and Borealis [7, 28]. Interest in SPEs led to work on performance optimization techniques such as operation scheduling [18, 25] and load shedding [35, 70]. We will revisit these works shortly (§3.4).

The second generation of "modern SPEs" began with Apache Storm [15] (2012) as part of the democratization of big data. Together with Apache Flink [13] and Twitter's Heron [49], these second-generation SPEs have been mainly developed by practitioners with a focus on scalable Cloud computing. They have achieved broad adoption in industry.

Under the hood, these modern SPEs are based on the One Worker Per Operation Architecture (OWPOA, Figure 2). In the OWPOA, the operations are connected by queues in a pipelined manner, and processed by its

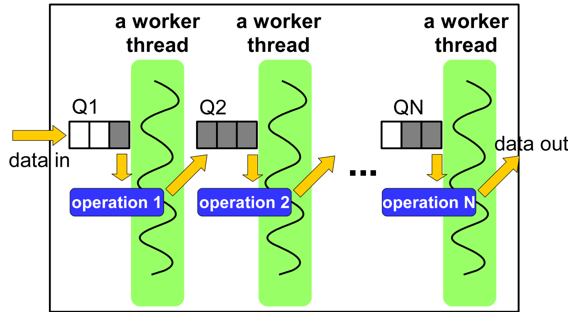


Figure 2: One Worker Per Operation Architecture (OWPOA). It assigns a worker thread for each operation. The example shows the case with `N` operations. `Q` represents a queue where a gray box means a pending data.

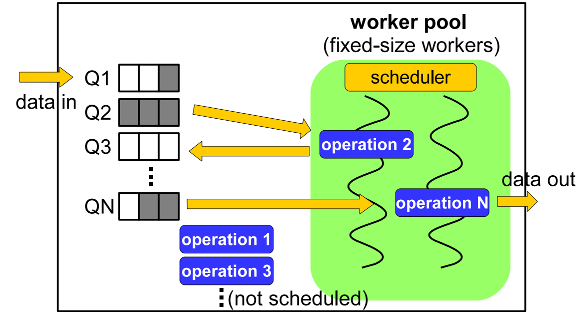


Figure 3: EDGEWISE Architecture (§4). The scheduler picks which operation to run. In this example, operation `2` and operation `N` had the longest queues, so they were scheduled.

own workers. Some operations may be mapped onto different nodes for distributed computing or to take advantage of heterogeneous resources (GPU, FPGA, etc.). In addition, the OWPOA monitors the health of the topology by checking the lengths of the per-operation queues. Queue lengths are bounded by a *backpressure* mechanism [32, 49], during which the source(s) buffer input until the operation queues clear.

3 Edge SPE Requirements Analysis

Stream processing is an important use case for Edge computing, but as we will show, existing SPEs are unsuited for the Edge. This section discusses our Edge SPE requirements analysis and the drawbacks of existing SPEs.

3.1 Our Edge Model

We first present our model for the Edge.

Hardware. Like existing IoT frameworks (e.g. Kura [36], EdgeX [2], and OpenFog [5]), we view the Edge as a distributed collection of IoT Gateways that connect Things to the Cloud. We consider Edge stream processing on IoT Gateways that are reasonably stable and well connected, unlike mobile drones or vehicles. We further assume these IoT Gateways have limited computing resources compared to the Cloud: few-core processors, little memory, and little permanent storage [17, 67]. However, they have more resources than those available to embedded, wireless sensor networks [50], and thus can afford reasonably complex software like SPEs. For example, Cisco's IoT Gateways come with a quad-core processor and 1GB of RAM [31].

Applications. Edge topologies consume IoT sensor data and apply a sequence of reasonably complex operations: e.g. SenML [44] parsers, Kalman filters, linear regressions, and decision tree classifications. For example, FarmBeats [72], a smart farm platform, uses IoT Gateways to collect data from various sensors and drones, to create summaries before sending them to the Cloud for long-term and cross-farm analytics, and to perform time-sensitive local data processing. We therefore assume a diverse set of workloads ranging from simple extraction-transform-load (ETL) and statistical summarization to predictive model training and classification.

3.2 Edge SPE Requirements

Every SPE aims for high throughput, and Edge SPEs are no exception. In addition, we see the following unique requirements for the Edge:

(1) Multiplexed. An Edge SPE must support an arbitrary topology on limited resources. We particularly focus on supporting a single topology in which there are more operations than processors, such that operation executions must be multiplexed on limited processors. Where a server-class machine can use threads unconcernedly, an Edge-class machine must be wary of unnecessary overheads.

(2) Low latency. An Edge SPE must offer low latency, else system architects should simply transmit raw data to the Cloud for analysis.

(3) No backpressure. The backpressure mechanism in Cloud SPEs is inadvisable at the Edge for two reasons. First, it is a "stop the world" scheme that destroys system latency, but latency is critical for Edge workloads. Second, it assumes that a data source can buffer pending data. While the Cloud can assume a persistent data source such as Kafka [4], at the Edge there is nowhere to put this data. Modern "real-time" SPEs such as Storm and Flink do not support file I/O based buffering. As a result the SPE must be congestion-aware to ensure queue lengths do not exceed available memory.

(4) Scalable. The SPE must permit scaling across multi-processors within an IoT Gateway and across multiple Gateways, especially to take advantage of locality or heterogeneity favorable to one operation or another.

3.3 Shortcomings of OWPOA-style SPEs

The OWPOA naturally takes advantage of intra-node and inter-node parallelism, especially when data engineers make a good logical-to-physical mapping. In the OWPOA, however, *multiplexing* becomes challenging in complex topologies, because each logical operation must have at least one worker and there may be more workers than cores on a compute node. If there are too many workers assigned to one compute node the poor scheduling of workers will artificially limit performance.

Let us explain the issue in detail. The OWPOA relies on the OS scheduler to decide which worker-operation pair to schedule next. If the input rate is low enough that most queues are empty (i.e. the SPE is over-provisioned), there is no harm in such an approach. Only some operations will have work to do and be scheduled, while the remainder will sleep.

But if the input rate rises (equivalently, if the SPE becomes less provisioned) then the SPE will become *saturated*, i.e. most or all queues will contain tuples. In this case any operation might be scheduled by the OS. As some operations take longer than others, a typical round-robin OS scheduler will naturally imbalance the queue lengths and periodically trigger backpressure. For example, in Figure 2, although Queue 2 is full, the OS may unwisely schedule the other worker-operation pair first, triggering *backpressure* unnecessarily and leading to significantly high *latency*.

3.4 A Lost Lesson: Operation Scheduling

As noted earlier (§2.2), the database community has studied different *profiling-guided* priority-based operation scheduling algorithms [18, 25] in the context of multiple operations and a single worker, before the modern OWPOA-style SPEs were born. For instance, Carney et al. [25] proposed a "Min-Latency" algorithm which assigns higher (static) priority on latter operations than earlier operations in a topology and processes old tuples in the middle of a topology before newly arrived tuples, with a goal to minimize average latency. For a topology with multiple paths, the tie is broken by the profiled execution time and input-output ratio of each operation. With a goal to minimize queue memory sizes, Babcock et al. [18] proposed a "Min-Memory" algorithm (called Chain) which favors operations with higher input-output reduction and short execution time (e.g., faster filters).

Unfortunately, however, modern OWPOA-style SPEs (e.g. Storm [15], Flink [13], Heron [49]) have not adopted these research findings when designing multi-core multi-worker SPEs and simply relied on the congestion-oblivious OS scheduler. *This paper argues that Edge SPEs should be rearchitected to regain the benefits of engine-level operation scheduling to optimize data flows in a multiplexed Edge environment.* In particular, we compare the effectiveness of both profiling-based (old) and dynamic balancing (new) scheduling algorithms, and report that all offer significant throughput and latency improvements over modern SPEs.

4 Design of EDGEWISE

This section presents EDGEWISE, an Edge-friendly SPE, that leverages a congestion-aware scheduler (§4.1) and a fixed-size worker pool (§4.2), as illustrated in Figure 3. EDGEWISE achieves higher throughput and low latency by *balancing the queue lengths*, with the effect of *pushing the backpressure point to a higher input rate*. Thus EDGEWISE achieves higher throughput without degrading latency (no backpressure). We later analyze the improved performance mathematically in §5.

4.1 Congestion-Aware Scheduler

EDGEWISE addresses the scheduling inefficiency of the OWPOA by incorporating a user-level scheduler to make wiser choices. In the OWPOA design, physical operations are coupled to worker threads and are scheduled according to the OS scheduler policy. The EDGEWISE scheduler separates the threads (execution) from the operations (data). Prior work has proposed using profiling-based operation scheduling algorithms [18, 25]. Instead, we propose a profiling-free dynamic approach that balances queue sizes by assigning a ready thread to the operation with the most pending data.

The intuition behind EDGEWISE is shown in Figure 4, which compares the behavior of an OWPOA-style SPE to EDGEWISE in a multiplexed environment. Figure 4(a) shows the unwise choice that may be made by the random scheduler of the OWPOA, leading to backpressure (high latency). Figure 4(b) contrasts this with the choice made by EDGEWISE's congestion-aware scheduler, evening out the queue lengths to avoid backpressure.

We believe EDGEWISE's congestion-aware scheduler would be beneficial to the Cloud context, as an intra-node optimization. In practice, however, we expect that EDGEWISE will have greater impact in the Edge setting, where (1) with few cores, there are likely more operators than cores, (2) latency is as critical as throughput, and (3) memory is limited.

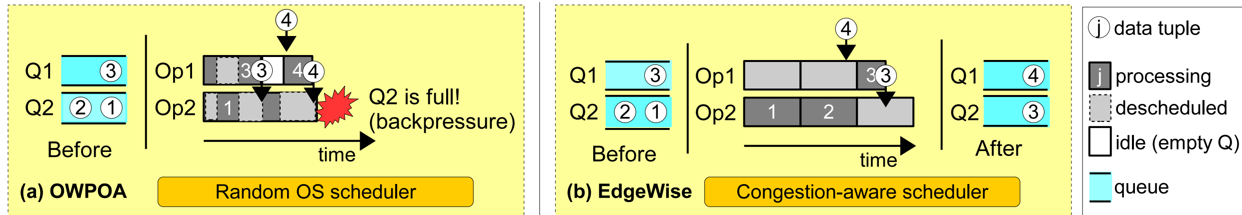


Figure 4: A scheduling example with two operations, multiplexed on a single core. The Q(ueue) size is two. Initially, Q1 has one tuple and Q2 is full. (a) The random OS scheduler in OWPOA lets Op(eration) 1 process tuples ③ and (newly coming) ④ first, overflowing Q2 and triggering unnecessary backpressure. (b) EDGEWISE knows Q2 has more pending data, and thus schedules the more critical Op2 first, avoiding congestion.

4.2 Fixed-size Worker Pool

EDGEWISE's scheduler decouples data from execution, requiring some changes in how EDGEWISE realizes the physical topology supplied by data engineers. Rather than dedicating a worker to each operation as in the OWPOA, EDGEWISE processes data on a fixed set of workers in the worker pool (Figure 3). These workers move from operation to operation as assigned by the scheduler. EDGEWISE could in principle work solely from the logical topology, in effect dynamically rearranging the physical topology (# workers assigned to each operation). However, this would violate any assumptions about thread safety or physical topology embedded in the logical topology. Thus, EDGEWISE uses the existing physical topology to bound the number of workers assigned to any operation at one time.

The worker pool size is configurable. By default it is set to the number of processors, because we assume that most operations are CPU-intensive per the recommended stream programming model (§2.1). If there are many I/O-bound operations, a fixed-size worker pool may lead to sub-optimal performance, requiring users to tune the pool size.

Putting it together. The scheduler dynamically chooses which operation a worker should perform. When a worker is ready, it asks the scheduler for an assignment, and the scheduler directs it to the operation with the longest pending data queue. Without work, it sleeps. When new data arrives, the scheduler wakes up the worker. The worker *non-preemptively*[1] completes the operation, and EDGEWISE directs the output to the downstream operation queue(s). Different operations may run in parallel: e.g., operations `2` and `N` in Figure 3. However, for FIFO guarantees, EDGEWISE schedules at most one worker to any operation instance (i.e. to any queue).

EDGEWISE supports three different data consumption policies for each scheduling turn: (1) `All` consumes all the tuples in the queue; (2) `Half` consumes half of the tuples in the queue; and (3) `At-most-N` consumes at most `N` tuples in the queue. Intuitively, in a saturated system, consuming a small constant number in each quantum will cause the scheduler to make more decisions overall. As our scheduler is fast, the more decisions it makes the better it should approximate the ideal schedule (evaluated in §7.4). Such consumption policies are not possible in OWPOA-style SPEs.

EDGEWISE meets the Edge SPE design goals, listed in §3.2. EDGEWISE retains the achievements of the OWPOA, and to these it adds the multiplexed, low-latency, and no-backpressure requirements. EDGEWISE's scheduler allows it to balance queue lengths, improving average latency and avoiding the need for backpressure by keeping heavier operations from lagging behind.

[1]More precisely, it is non-preemptive at the engine level. EDGEWISE can still be preempted by the underlying OS scheduler.

5 Performance Analysis of EDGEWISE

This section shows analytically that EDGEWISE will achieve higher throughput (§5.1) and lower latency (§5.2) than OWPOA. Lastly, in §5.3 we discuss how to measure relevant runtime metrics for the performance analysis. To the best of our knowledge, we are the first to apply queueing theory to analyze the improved performance in the context of stream processing. Prior scheduling works in stream processing either provide no analysis [25] or focus only on memory optimization [18].

5.1 Higher Throughput

First we show that maximum end-to-end throughput depends on scheduling heavier operations proportionally more than lighter operations. This is impossible to guarantee in the scheduler-less modern SPEs, but easy for EDGEWISE's scheduler to accomplish.

Our analysis interprets a dataflow topology as a queuing network [38, 46]: a directed acyclic graph of stations. Widgets (tuples) enter the network via the queue of the first station. Once the widget reaches the front of a station's queue, a server (worker) operates on it, and then it advances to the next station. The queuing theory model

allows us to capture the essential differences between EDGEWISE and the OWPOA. In the rest of this section we draw heavily on [38, 46], and we discuss the model's deviations from real topology behavior at the end.

Modeling. Given the input rate λ_i and the service rate μ_i of a server i, the server utilization ρ_i (fraction of time it is busy) is defined as:

$$\rho_i = \frac{\lambda_i}{\mu_i} \tag{1}$$

A queuing network is *stable* when $\rho_i < 1$ for all servers i. If there is a station to which widgets arrive more quickly than they are serviced, the queue of that station will grow unbounded. In an SPE, unbounded queue growth triggers the backpressure mechanism, impacting system latency.

Suppose we want to model a topology with M operations in this way. We can represent the input and server rates of each operation as a function of λ_0 and μ_0, the input and service rates of the first operation, respectively. We are particularly interested in λ_0 as it is the input rate to the system; higher λ_0 means higher throughput. Expressing the input scaling factors and relative operation costs for an operation i as q_i and r_i, we can write the input rate λ_i and the service rate μ_i for operation i as:

$$\lambda_i = q_i \cdot \lambda_0 \qquad \mu_i = r_i \cdot \mu_0$$

In queuing theory, each node is assumed to have an exclusive server (worker). However, in the Edge, there may be fewer servers (processor cores) than stations (operations). If there are C processor cores, we can model processor contention by introducing a scheduling weight w_i, subject to $\sum_i^M w_i = C$, yielding the effective service rate $\mu_i{}'$:

$$\mu_i{}' = w_i \cdot \mu_i = w_i \cdot (r_i \cdot \mu_0)$$

For instance, when one core is shared by two operations, a fair scheduler halves the service rate ($w_1 = w_2 = \frac{1}{2}$) as the execution time doubles. The constraint $\sum_i^M w_i = C$ reflects the fact that C processor cores are shared by M operations.

Similarly, the effective server utilization $\rho_i{}'$ can be redefined based on $\mu_i{}'$. To keep the system stable (eliminate backpressure), we want $\rho_i{}'$ to be less than one:

$$\forall i, \quad \rho_i{}' = \frac{\lambda_i}{\mu_i{}'} = \frac{q_i \cdot \lambda_0}{w_i \cdot r_i \cdot \mu_0} < 1 \tag{2}$$

which can be rearranged as

$$\forall i, \quad \lambda_0 < w_i \cdot \frac{r_i}{q_i} \cdot \mu_0 \tag{3}$$

Optimizing. Remember, λ_0 represents the input rate to the system and is under our control; a higher input rate means higher throughput (e.g. the system can sample sensors at a higher rate). Assuming that μ_0, r_i, and q_i are constants for a given topology, the constraint of Equation (3) implies that the maximum input rate λ_0 that an SPE can achieve is upper-bounded by the minimum of $w_i \cdot \frac{r_i}{q_i}$ over all i. Thus, our optimization objective is to:

$$\begin{aligned} &\text{maximize} \quad \min_i (w_i \cdot \frac{r_i}{q_i}) \\ &\text{subject to} \quad \sum_i^M w_i = C \end{aligned} \tag{4}$$

It is straightforward to show that the maximum input rate can be achieved when $w_i \cdot \frac{r_i}{q_i}$ are equal to each other for all i: i.e.,

$$w_1 : w_2 : \cdots : w_N = \frac{q_1}{r_1} : \frac{q_2}{r_2} : \cdots : \frac{q_M}{r_M}$$

In other words, the scheduling weight w_i should be assigned proportional to $\frac{q_i}{r_i}$. This is indeed very intuitive. An operation becomes heavy-loaded when it has a higher input rate (higher q_i) and/or a lower service rate (lower r_i). Such an operation should get more turns.

Scheduling. EDGEWISE's scheduler dynamically identifies heavily loaded operations and gives them more turns. The queue of an operation with higher input rate and higher compute time (lower service rate) grows faster than the others. By monitoring queue sizes, we can identify and favor the heavy-loaded operations with more frequent worker assignments.

Compare this behavior to that of the OWPOA. There, the "fair" OS scheduler blindly ensures that w_i is the same for each operation, unaware of relative input and service rates, leading to suboptimal throughput. Once saturated, one of the heavy operations will reach the utilization cap of $\rho_i = 1$, become a bottleneck in the topology, and eventually trigger backpressure and latency collapse. Data engineers can attempt to account for this by profiling their topologies and specifying more workers for heavier operations. Fundamentally, however, this is an ad hoc solution: the OS scheduler remains blind.

In our evaluation (§7.3.1) we show that EDGEWISE achieves $w_i \cdot \frac{r_i}{q_i}$ equality across operations at runtime, leading to an optimal balanced effective server utilization $\rho_i{}'$. In contrast, we report that in the OWPOA approach, increasing the input rate leads to increasingly unbalanced $\rho_i{}'$ across operations.

Runtime deviations from the model. Our queuing theory model captures real SPE behavior in the essentials, but it deviates somewhat in the particulars. We give two prominent examples. The first deviation is that our queuing theory model assumes predictable widget fan-in and fan-out ratios at each operator (i.e. constant q_i and r_i). In reality these are distributions. For example, an operation that splits a sentence into its constituent words

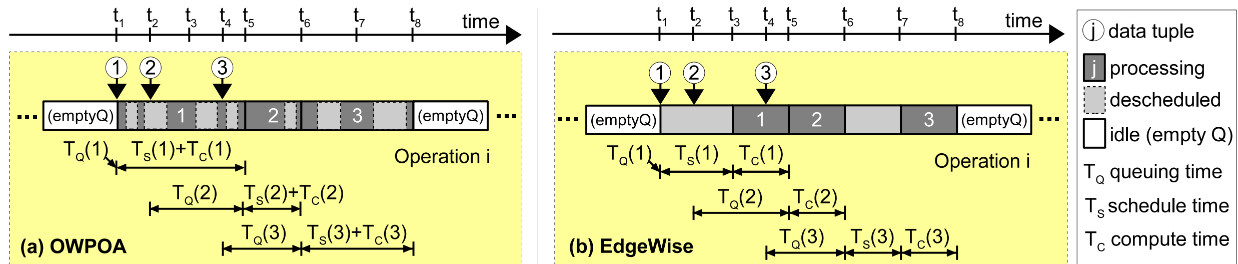


Figure 5: A per-operation latency breakdown example of (a) OWPOA, and (b) EDGEWISE. Incoming data tuples ①, ②, and ③ are being queued. When scheduled (dark gray box), the operation *i* processes a pending tuple. A tuple *j* can be in one of three states: waiting in the middle of the queue (T_Q), at the head of the queue waiting to be scheduled (T_S), or being computed (T_C). Later in §7.3.2, we show that T_Q dominates the per-operation latency in a saturated system.

will emit one tuple for "Please", two tuples for "Please accept", and so on. The second deviation is that we assumed that operations have constant costs. In reality it will take such a sentence-splitting operation more time to process longer sentences.

Considering these deviations, the difficulty data engineers face when generating physical topologies for a OWPOA SPE is clear: it is difficult to identify a single integer quantity of workers to assign to each operation, and even then balance will only be achieved probabilistically over many scheduling cycles. In contrast, EDGEWISE automatically balances the assignment of workers to operations.

5.2 Lower Latency

Now we show that EDGEWISE can achieve lower end-to-end *latency* than the OWPOA without compromising throughput. Our analysis hinges on the observation that unbalanced queue lengths have an outsized effect on latency, so that balancing queue lengths leads to an overall improvement in latency.

The total latency for a completed tuple equals the sum of the latencies paid at each operation it visited on its way from source to sink, plus communication latencies outside of our control:

$$Latency = \sum_{i}^{M}(L_i + Comm.) \approx \sum_{i}^{M} L_i \quad (5)$$

In stream processing, the per-operation latency consists of (1) the queue time T_Q, waiting in the queue; (2) the schedule time T_S, waiting for a turn at the head of the queue; and (3) the (pure) compute time T_C, being processed. As illustrated in Figure 5, these times come at a different form. In OWPOA (a), the preemptive OS scheduler makes T_S and T_C interleaved. In EDGEWISE (b), the non-preemptive scheduler makes clear distinction because an operation is not scheduled until a worker is available, and when scheduled, it completes its work.

In a saturated system, we treat T_S and T_C as constants, while T_Q will grow with the input rate to dominate L_i.

$$L_i = T_Q + (T_S + T_C) \approx T_Q \quad (6)$$

Assuming that the input and service rates, λ and μ, can be modeled as exponential random variables[2], Gross et al. [38] show that the queue time T_Q can be expressed as

$$T_Q = \frac{\rho}{\mu - \lambda} = \frac{\lambda}{\mu(\mu - \lambda)} \quad (7)$$

Note that T_Q has a vertical asymptote (approaches ∞) at $\mu = \lambda$, and a horizontal asymptote (approaches 0) when $\mu >> \lambda$. In other words, for a given λ, a tuple will wait longer in the queues of heavier operations, and crucially *the growth in wait time is non-linear (accelerates) as* μ *approaches* λ. This means that heavier operations have a much larger T_Q and thus L_i (Equation (6)) than lighter operations, and an outsized impact on overall latency (Equation (5)).

Though the effect of heavy operations may be dire when using the OWPOA's random scheduler, EDGEWISE's congestion-aware scheduler can avoid this problem by giving more hardware resources (CPU turns) to heavier operations over lighter ones. In effect, EDGEWISE balances the T_Q of different operations, reducing the T_Q of tuples waiting at heavy operations but increasing the T_Q of tuples waiting at lighter operations. According to Gross et al.'s model this balancing act is not a zero-sum game: we expect that the lighter operations sit near the horizontal asymptote while the heavier operations sit near the vertical asymptote, and balancing queue times will shift the entire latency curve towards the horizontal asymptote. In our evaluation we support this analysis empirically (§7.3.2).

[2] For λ and μ with general distributions, the curve in the $\lambda < \mu$ region is similar. The exponential model simplifies the presentation.

5.3 Measuring Operation Utilization

In our evaluation we compare EDGEWISE with a state-of-the-art OWPOA. For a fine-grained comparison, in addition to throughput and latency we must compare the operation utilization $\rho = \frac{\lambda}{\mu}$ discussed in §5.1. This section describes how we can fairly compare EDGEWISE against a baseline OWPOA system in this regard.

We want to compare the server (operation) utilization of EDGEWISE and the baseline. As argued in §5.1, an ideal result is a balanced utilization vector consisting of equal values $1-\varepsilon$, with ε chosen based on latency requirements. As operation utilization may change over the lifetime of an SPE run (e.g. as it reaches "steady-state"), we incorporate a time window T_w into the metric given in Equation (1).

During a time window T_w, an operation might be performed on N tuples requiring a total of T_E CPU time. The input rate for this operation during this window is $\lambda_w = \frac{N}{T_w}$ and the service rate is $\mu_w = \frac{1}{T_E}$, so we have windowed utilization ρ_w as:

$$\rho_w = \frac{\lambda_w}{\mu_w} = T_E \cdot \frac{N}{T_w} \tag{8}$$

Unsurprisingly, we found that this is the utilization metric built into Storm [15][3], the baseline system in our evaluation.

In the OWPOA, computing ρ_w is easy. T_w is fixed, N is readily obtained, and T_E can be calculated by monitoring the beginning and ending time of performing the operation on each tuple:

$$T_E = \frac{1}{N}\sum_{j}^{N}(T_{end}(j) - T_{begin}(j)) \tag{9}$$

Equation (9) would suffice to measure ρ_w for the OWPOA, but for a fair comparison between the OWPOA and EDGEWISE we need a different definition. Note that $T_{end}(j) - T_{begin}(j)$ captures the total time a worker spends applying an operation to a tuple, and in the OWPOA (Figure 5 (a)) this calculation includes both T_C (pure computation) and T_S (worker contention). As EDGEWISE's scheduler is non-preemptive (Figure 5 (b)), measuring T_E in this way would capture only EDGEWISE's T_C. Doing so would ignore its T_S, the time during which EDGEWISE's scheduler decides *not* to schedule an operation with a non-empty queue in favor of another operation with a longer queue. This would artificially decrease T_E and thus ρ_w for this operation.

To capture T_S for an operation in EDGEWISE, we can instead amortize the schedule time T_S across all completed tuples, and describe the operation's time during this window as spent either executing or idling with an empty queue:

$$T_w = T_E \cdot N + T_{emptyQ} \tag{10}$$

Solving Equation (10) for T_E and substituting into Equation (8), we can compute ρ_w in EDGEWISE:

$$\rho_w = 1 - \frac{T_{emptyQ}}{T_w} \tag{11}$$

The windowed utilization metric ρ_w given in Equation (11) applies equally well to SPEs that use the OWPOA or EDGEWISE.

[3] In Apache Storm this metric is called the operation's *capacity*.

6 Implementation

We implemented EDGEWISE on top of Apache Storm [15] (v1.1.0). Among modern SPEs, Storm was the most popular for data science in 2018 [20] and has the lowest overall latency [29]. We made three major modifications: (1) We implemented the congestion-aware scheduler (§4.1), and added two data structures: a list of operations with non-empty queues as scheduling candidates; and a list of running operations to ensure FIFO per queue; (2) We removed the per-operation worker threads, and added one worker pool of (configurable) K worker threads (§4.2); and (3) We introduced two queuing-related metrics, T_{emptyQ} and T_Q, for server utilization and latency breakdown analysis (§5.3).

EDGEWISE can be applied to other OWPOA-style SPEs such as Flink [13] and Heron [49]. In Flink, multiple operators may be grouped in a single Task Slot, but each operator (called "subtask") still has its own worker thread. Task Slot separates only the managed "memory" of tasks, but there is no CPU isolation. As a result, worker threads will still contend and cause congestion if there are more operators than CPUs. Thus, EDGEWISE's congestion-aware scheduler and fixed-size worker pool will be equally beneficial for Flink.

The EDGEWISE prototype is available at `https://github.com/VTLeeLab/EdgeWise-ATC-19`. It adds 1500 lines of Java and Clojure across 30 files.

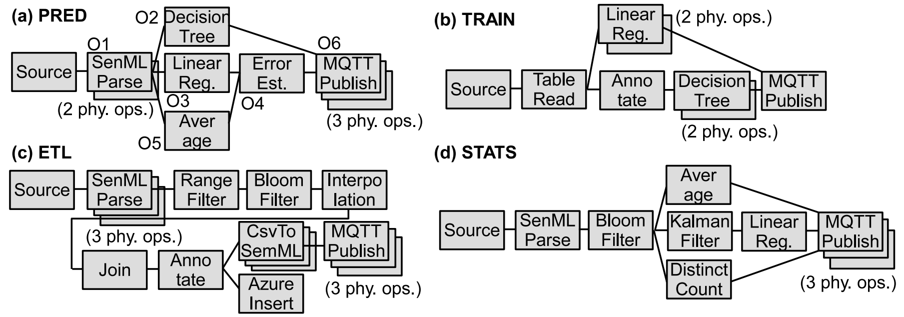
7 Evaluation

Our evaluation first shows EDGEWISE's throughput-latency performance on representative Edge stream processing workloads (§7.2), followed by detailed performance analysis (§7.3). Then we present a sensitivity study on different consumption policies (§7.4), and a distributed (inter-node) performance study (§7.5).

7.1 General Methodology

Hardware. EDGEWISE is designed for the Edge, so experiments use an intermediate-class computing device

Figure 6: Tested Topologies in RIoTBench [68]: (a) PREDictive analytics, (b) model TRAINinig, (c) Extract, Transform, and Load, and (d) STATistical summarization. The (tuned) number of physical operations are shown as the overlapped, multiple rectangles: e.g., 3 MQTT Publish operations in (a).



representative of IoT Edge devices (§3.1): weaker than Cloud-class servers but stronger than typical embedded systems. Specifically, we use Raspberry Pi 3 Model B devices (raspbis), which have a 1.2GHz quad-core ARM Cortex-A53 with 1GB of RAM and run Raspbian GNU/Linux 8.0 v4.1.18.

Baseline. We used Storm [15] as the OWPOA baseline. We set up Storm's dependencies, Nimbus and a Zookeeper server, on a desktop machine, and placed the Storm supervisors (compute nodes) on the raspbis. Because EDGEWISE optimizes the performance of a single compute node, all but our distributed experiment (§7.5) use a single raspbi. As our raspbis are quad-core, EDGEWISE uses four worker threads.

Schedulers. In addition to EDGEWISE's queue-length-based approach, we also evaluated implementations of the Min-Memory [18] and Min-Latency [25] schedulers, as well as a Random scheduler. All schedulers used our `At-most-50` data consumption policy (§4.2), based on our sensitivity study §7.4.

Benchmarks. We used the RIoTBench benchmark suite [68], a real-time IoT stream processing benchmark implemented for Storm[4]. The RIoTBench benchmarks perform various analyses on a real-world Smart Cities data stream [22]. Each input tuple is 380 bytes. The sizes of intermediate tuples vary along the topology.

Figure 6 shows RIoTBench's four topologies. We identified each physical topology using a search guided by the server utilization metric (§5.1), resulting in physical topologies with optimal latency-throughput curves on our raspbi nodes [40]. We used the same physical topologies for both Storm and EDGEWISE.

We made three modifications to the RIoTBench benchmarks: (1) We patched various bugs and inefficiencies. (2) We replaced any Cloud-based services with lab-based ones; (3) To enable a controlled experiment, we implemented a timer-based input generator that reads data from a replayed trace at a configurable input rate. To be more specific, the Smart Cities data [22] is first loaded into the memory, and a timer periodically (every 100 ms) feeds a fixed-size batch of them into `Spout`, the source operator in Storm. We changed the batch size to vary the input rate. This simulates a topology measuring sensor data at different frequencies, or measuring different number of sensors at a fixed frequency.

Metrics. Measurements were taken during the one minute of steady-state runs, after discarding couple minutes of initial phase. We measured *throughput* by counting the number of tuples that reach the `MQTT Publish` sink. Measuring throughput at the sink results in different throughput rates for each topology at a given input rate, since different topologies have input-output tuple ratios: e.g., 1:2 in `PRED`, 1:5 in `STATS`. We measured *latency* by sampling 5% of the tuples, assigning each tuple a unique ID and comparing timestamps at source and the same sink used for the throughput measurement. We measured *operation utilization* using ρ_w (Equation (11)). Each experiment was performed 5 times with each configuration. The error bars indicate one standard deviation from the average. Most data points had small variances.

7.2 Throughput-Latency Performance

We measured the throughput-latency performance curve for each of the RIoTBench applications on one raspbi across a range of input rates. The curves for `PRED`, `STATS`, and `ETL` are shown in Figure 7; the curve for the `TRAIN` application (not shown) looks like that of the `ETL` application. In general, both Storm and EDGEWISE have excellent performance when the system is under-utilized (low throughput). Latency performance collapses at high throughput rates as a result of frequent backpressure.

The results show that an SPE with an engine-level operation scheduler and a worker pool (WP) significantly outperforms Storm (an OWPOA-based SPE) at the Edge, in effect shifting the Storm throughput-latency curve down (lower latency) and to the right (higher throughput). First, the gaps between Storm and WP+Random indicate

[4] Other benchmarks [6, 53] seemed too unrealistic for our use case.

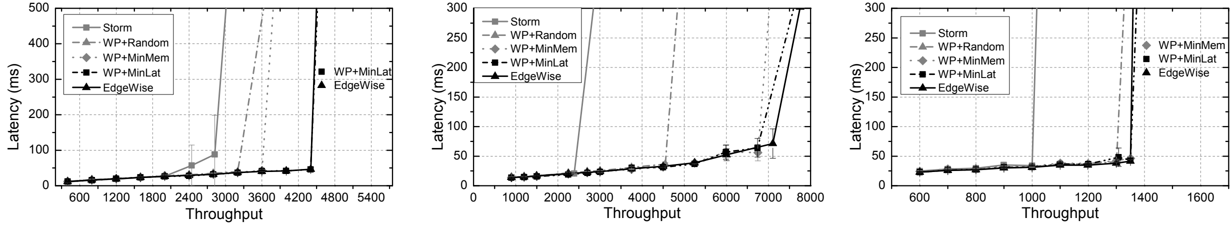


Figure 7: Throughput-latency of (a) `PRED`, (b) `STATS`, and (c) `ETL` topologies.

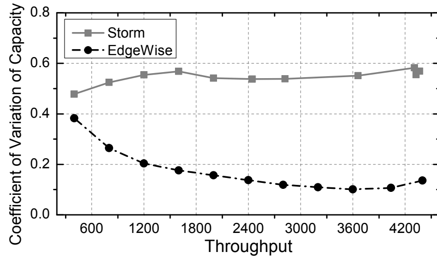


Figure 8: In `PRED`, as the input rate (thus throughput) increases, the coefficient of variation (CV) of capacities grows in Storm, but it decreases in EDGEWISE.

the benefit of avoiding unnecessary contentions in OW-POA. More importantly, the engine-level schedulers in effect push the backpressure point to a higher input rate, allowing the SPEs to achieve higher throughput at a low latency. The high variance in the Storm curves at higher throughput rates indicate early, random backpressure.

Among the variants that use a scheduler and WP, EDGEWISE's queue-length-based scheduler matches or outperforms WP+MinLat on latency and throughput, while WP+MinMem leads to improved (but not the best) performance as it is optimized for memory. Note that EDGEWISE does not require profiling per-operation execution time and input-output ratio as do MinLat and MinMem. For `PRED`, while keeping the latency low (say $\leq$100 ms), EDGEWISE improves its throughput by 57% (from 2800 to 4400). EDGEWISE is particularly effective in the `STATS`, where it achieves a 3x throughput improvement with low latency. For `ETL`, EDGEWISE improves throughput from 1000 to 1350 under 50ms latency.

7.3 Detailed Performance Breakdown

This experiment investigates the underlying causes of the throughput and latency gains described in §7.2, lending empirical support to our analysis in §5. We use the `PRED` application as a case study, though results were similar in the other RIoTBench applications.

7.3.1 Fine-Grained Throughput Analysis

Our analysis of throughput in §5.1 predicted that balancing effective server (operation) utilization would yield gains in throughput. To measure the extent to which Storm and EDGEWISE balance server utilization, in this experiment we calculated the windowed utilization ρ_w of each operation using Equation (11) and then computed the coefficient of variation ($CV = \frac{stddev}{avg}$) of this vector. A lower utilization CV means more balanced server utilization.

Figure 8 plots the coefficient of variation for Storm and EDGEWISE for different input rates. As the input rate (and thus output throughput) increases, the utilization CV increases in Storm, indicating that the operations become unbalanced as Storm becomes saturated. In contrast, in EDGEWISE the utilization CV decreases for larger input rates (and the raw ρ_w values approach 1). As predicted, EDGEWISE's throughput gains are explained by its superior ρ_w balancing.

7.3.2 Fine-Grained Latency Analysis

Our analysis of latency in §5.2 noted that in a congestion-blind scheduler, heavier operations would develop longer queues, and that the queuing time at these operations would dominate end-to-end latency. We showed that an SPE that reduced queuing times at heavy operations, even at the cost of increased queuing times at lighter operations, would obtain an outsized improvement in latency. In this experiment we validate this analysis empirically.

For this experiment we break down per-operation latency into its constituent parts (Equation (6)) and analyze the results in light of our analysis.

Figure 9 shows the per-operation latency for the 6 non-source operations in the `PRED` topology in the baseline Storm. Its logical and physical topology is shown in Figure 6 (a). Where our physical topology has multiple instances of an operation (two instances of `O1`, three of `O6`), we show the average queuing latency across instances.

Note first that the queue time T_Q dominates the per-operation latency L_i. Then, as the input rate increases from L(ow) to H(igh), the queues of heavier operations (`O1`, `O6`) grow much longer than those of lighter operations, and the queue time at these operations dominates the overall latency. Also note that the latency of lighter operations may decrease at high throughput, as tuples are

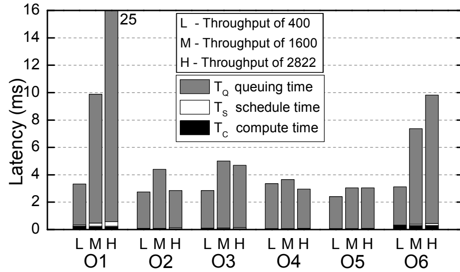


Figure 9: In Storm, as the throughput increases from L(ow) to M(edium) to H(igh), the queuing latency of heavy operations (e.g., `O1` and `O6`) increases rapidly.

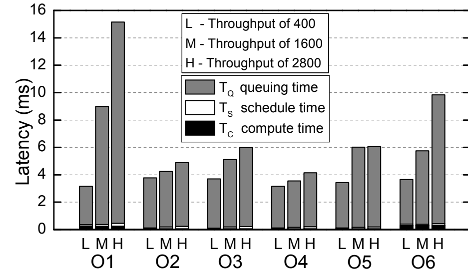


Figure 10: In EDGEWISE, as the throughput increases, the queuing latency of heavy operations (e.g., `O1` and `O6`) increases slowly.

mostly waiting in the queue of heavy operations, reflecting the effects of the non-deterministic OS scheduler.

In contrast, we observed different behavior for EDGEWISE as shown in Figure 10. The heaviest operation `O1` is still noticeable but its queue time under High input rate is only 15 ms, much smaller than the 25 ms of Storm. Of course this penalizes the lighter operations, but as we argued in §5.2 this is not a zero-sum game; the improvement in end-to-end latency outweighs any small per-operation latency increase.

The schedule time T_S includes both the scheduling overhead and the waiting time due to contention. Across all throughput rates and operations, T_S remains very small, implying that the overhead of EDGEWISE's scheduler is negligible.

7.4 Data Consumption Policy

In this experiment we explored the sensitivity of EDGEWISE's performance to its data consumption policies: a constant number (`At-most-N`) or a number proportional to the queue length (`All`, `Half`).

In Figure 11 you can see the effect of these rules in the `STATS` topology, as well as the Storm performance for comparison. As expected, the constant consumption rules consistently performed well in `STATS`. The `PRED` showed the trend similar to the `STATS`. The `TRAIN` and `ETL` topologies were not sensitive to the consumption policy. The `At-most-50` rule (solid black line) offers good latency with the highest throughput for all, so we used it in our other experiments.

7.5 Performance on Distributed Edge

Streaming workloads can benefit from scaling to multiple compute nodes, and supporting scaling was one of our design goals (§3.2). In this experiment we show that EDGEWISE's intra-node optimizations benefit an inter-node (distributed) workload.

We deployed the `PRED` application across 2, 4, and 8 raspbis connected on a 1G Ethernet lab network, simply

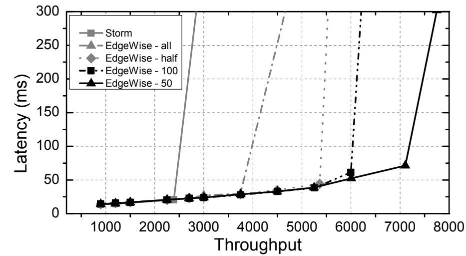


Figure 11: Sensitivity study on various consumption policies with `STATS` topology.

increasing the physical operation instances proportional to the number of raspbis and assigning them uniformly across nodes. Identifying an optimal distributed topology (and optimal partitioning across nodes) is out of scope for our work, which focuses on the optimal scheduling of the topology deployed on a single node. Experiments on visionary hundred- or thousand-node cases are left for future work. We used the same methodology as in §7.2 to collect metrics.

As expected, EDGEWISE's intra-node optimizations prove beneficial in an inter-node setting. Figure 12 shows the maximum throughput achieved by Storm and EDGEWISE with latency less than 100 ms. The scalability curve shows about 2.5x throughput improvement with 8 nodes, suggesting that a combination of networking costs and perhaps a non-optimal topology keep us from realizing the full 8x potential improvement. On this particular physical topology, EDGEWISE achieves an 18–71% improvement over Storm's maximum throughput, comparable to the 57% improvement obtained in the 1-node experiment.

8 Related Work

To the best of our knowledge, Edgent [12] is the only other SPE tailored for the Edge. Edgent is designed for data preprocessing at individual Edge devices rather than

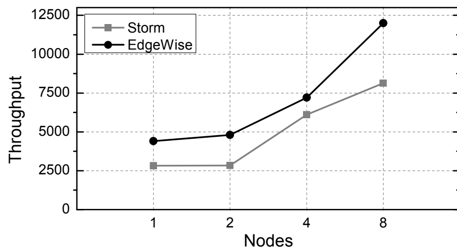


Figure 12: The maximum throughput achieved with the latency less than 100 ms, using the PRED topology with 1, 2, 4, and 8 distributed nodes (log-scale). EDGEWISE's intra-node optimizations extend to a distributed setting.

full-fledged distributed stream processing. We believe the Edge is powerful enough for more intelligent services, and thus EDGEWISE targets a more expressive SPE language with balanced throughput and latency.

Targeting distributed Cloud settings, a variety of academic [26, 39, 52, 62, 73] and industry/open-source [15, 49, 66] SPEs have been proposed. Some [8, 9, 11, 13, 14] support both stream and batch processing. Most use the OWPOA design and differentiate themselves on large-scale distributed processing and fault tolerance. None has our notion of an engine-level scheduler.

Researchers have studied distributed "job placement" schedulers for Storm [10, 23, 60, 77] and Spark Streaming [47, 51]. They determine where to distribute computing workloads across nodes in the Cloud. EDGEWISE does not focus on how to partition a physical topology for distributed computing.

Recent single-node SPE solutions leverage modern many-core and heterogeneous architectures. For example, GStream [78] describes an SPE tailored to GPUs, while Saber [48] is a hybrid SPE for mixed CPU/GPU computing that schedules operations on different hardware depending on where they perform better. StreamBox [56] explores the high-end server space, demonstrating high throughput by extracting pipeline parallelism on a state-of-the-art 56-core NUMA server. StreamBox uses a worker pool (like EDGEWISE) to maximize CPU utilization, but does not maintain a queue for each operation, making it hard to apply the Storm-like distributed stream processing model. EDGEWISE targets the opposite end of the spectrum: a compute cluster composed of Edge-class devices where intra-node scale-up is limited but inter-node scale-out is feasible. Thus, EDGEWISE adopts the general distributed streaming model (scale-out) and enables additional intra-node scale-up.

The Staged Event Driven Architecture (SEDA) [75] was proposed to overcome the limitations of thread-based web server architectures (e.g., Apache Httpd [1]): namely, per-client memory and context-switch overheads. EDGEWISE shares the same observation and proposes a new congestion-aware scheduler towards a more efficient EDA, considering how to stage (schedule) stream processing operations.

Mobile Edge Computing (MEC) [42, 59] uses mobile devices to form an Edge. Unlike EDGEWISE, they focus on mobile-specific issues: e.g., mobility, LTE connections, etc. Two works support stream processing on mobile devices. Mobile Storm [57] ported Apache Storm as is. MobiStreams [74] focuses on fault tolerance when a participating mobile disappears. On the other hand, Mobile-Cloud Computing (MCC) [30, 34, 37, 45, 63] aims to offload computation from mobile to the Cloud. There will be a good synergy between EDGEWISE and MEC/MCC. EDGEWISE could be viewed as a local cloud to which they can offload computations.

Lastly, queueing theory has been used to analyze stream processing. The chief difference between previous analyses and our own lies in the assumption about worker sharing. In traditional queueing theory, each operation is assumed to fairly share the workers. For example, Vakilinia et al. [71] uses queueing network models to determine the smallest number of workers under the latency constraint, under the assumption that workers are uniformly shared among operations. The same is true for the analyses of Mak et al. [54] for series-parallel DAG and Beard et al. [19] for heterogeneous hardware. On the other hand, our analysis identifies the benefit of a non-uniform scheduling weight, assigning more workers to heavy-loaded operations.

9 Conclusion

Existing stream processing engines were designed for the Cloud and behave poorly in the Edge context. This paper presents EDGEWISE, a novel Edge-friendly stream processing engine. EDGEWISE improves throughput and latency thanks to its use of a congestion-aware scheduler and a fixed-size worker pool. Some of the ideas behind EDGEWISE were proposed in the past but forgotten by modern stream processing engines; we enhance these ideas with a new scheduling algorithm supported by a new queuing-theoretic analysis. Sometimes the answers in system design lie not in the future but in the past.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Dilma Da Silva, for their thoughts and guidance. This work was supported in part by the National Science Foundation, under grant CNS-1814430.

References

[1] The apache http server. http://httpd.apache.org.

[2] EdgeX Foundry. https://www.edgexfoundry.org/.

[3] Ispout api documentation. https://storm.apache.org/releases/1.1.2/javadocs/org/apache/storm/spout/ISpout.html.

[4] Kafka - A distributed Streaming Platform. https://kafka.apache.org/.

[5] OpenFog. https://www.openfogconsortium.org/.

[6] Storm benchmark. https://github.com/intel-hadoop/storm-benchmark.

[7] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[9] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[10] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[11] Apache. Apache beam. https://flink.apache.org/.

[12] Apache. Apache Edgent - A Community for Accelerating Analytics at the Edge. https://edgent.apache.org/.

[13] Apache. Apache flink. https://flink.apache.org/.

[14] Apache. Apache spark. a fast and general engine for large-scale data processing. http://spark.apache.org/.

[15] Apache. Apache storm. an open source distributed realtime computation system. http://storm.apache.org/.

[16] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[18] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, 13(4):333–353, 2004.

[19] Jonathan C Beard and Roger D Chamberlain. Analysis of a simple approach to modeling performance for streaming data applications. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349. IEEE, 2013.

[20] Sean Boland. Ranking popular distributed computing packages for data science, 2018.

[21] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, 2012.

[22] Data Canvas. Sense your city: Data art challenge. http://datacanvas.org/sense-your-city/.

[23] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[24] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the*

28th international conference on Very Large Data Bases, pages 215–226. VLDB Endowment, 2002.

[25] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 838–849. VLDB Endowment, 2003.

[26] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[27] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[28] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[29] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[30] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[31] Cisco. Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[32] Rebecca L Collins and Luca P Carloni. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 205–214. ACM, 2009.

[33] Mckinsey & Company. The internet of things: Mapping the value beyond the hype, 2015.

[34] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[35] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 40–51, New York, NY, USA, 2003. ACM.

[36] Eclipse. Eclipse kura. open-source framework for iot. http://www.eclipse.org/kura/.

[37] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[38] Donald Gross. *Fundamentals of queueing theory*. John Wiley & Sons, 2008.

[39] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[40] HortonWorks. Apache storm topology tuning approach. https://community.hortonworks.com/articles/62852/feed-the-hungry-squirrel-series-storm-topology-tun.html.

[41] HortonWorks. Hortonworks best practices guide for apache storm. https://community.hortonworks.com/articles/550/unofficial-storm-and-kafka-best-practices-guide.html.

[42] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11(11):1–16, 2015.

[43] International Electrotechnical Commission (IEC). Iot 2020: Smart and secure iot platform. http://www.iec.ch/whitepaper/pdf/iecWP-loT2020-LR.pdf.

[44] C. Jennings, Z. Shelby, J. Arkko, and A. Keranen. Media types for sensor markup language (senml). 2016.

[45] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 615–629, New York, NY, USA, 2017. ACM.

[46] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[47] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Dens: Data center energy-efficient network-aware scheduling. *Cluster Computing*, 16(1):65–75, March 2013.

[48] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[49] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[50] Franck L Lewis et al. Wireless sensor networks. *Smart environments: technologies, protocols, and applications*, 11:46, 2004.

[51] Zhen Li, Bin Chen, Xiaocheng Liu, Dandan Ning, Qihang Wei, Yiping Wang, and Xiaogang Qiu. Bandwidth-guaranteed resource allocation and scheduling for parallel jobs in cloud data center. *Symmetry*, 10(5):134, 2018.

[52] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, volume 16, pages 439–453, 2016.

[53] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 69–78, Washington, DC, USA, 2014. IEEE Computer Society.

[54] Victor W Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, 1990.

[55] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 215–224, New York, NY, USA, 1989. ACM.

[56] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. Streambox: Modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 617–629, Berkeley, CA, USA, 2017. USENIX Association.

[57] Qian Ning, Chien-An Chen, Radu Stoleru, and Congcong Chen. Mobile storm: Distributed real-time stream processing for mobile clouds. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 139–145. IEEE, 2015.

[58] A. Papageorgiou, E. Poormohammady, and B. Cheng. Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 259–266, June 2016.

[59] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. Mobile-edge computing

introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[60] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[61] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304. ACM, 2011.

[62] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.

[63] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[64] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178, Oct 2016.

[65] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[66] Scott Schneider and Kun-Lung Wu. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. *ACM SIGPLAN Notices*, 52(6):648–661, 2017.

[67] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[68] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: a real-time iot benchmark for distributed stream processing platforms. *arXiv preprint arXiv:1701.08530*, 2017.

[69] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[70] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[71] Shahin Vakilinia, Xinyao Zhang, and Dongyu Qiu. Analysis and optimization of big-data stream processing. In *2016 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[72] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *NSDI*, pages 515–529, 2017.

[73] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[74] Huayong Wang and Li-Shiuan Peh. Mobistreams: A reliable distributed stream processing system for mobile devices. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 51–60. IEEE, 2014.

[75] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[76] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[77] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544. IEEE, 2014.

[78] Yongpeng Zhang and Frank Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 245–254. IEEE, 2011.

Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads

Myeongjae Jeon[†*], Shivaram Venkataraman[‡*], Amar Phanishayee[*], Junjie Qian[*], Wencong Xiao[§*], and Fan Yang[*]

[†]*UNIST* [‡]*University of Wisconsin* [§]*Beihang University* [*]*Microsoft Research*

Abstract

With widespread advances in machine learning, a number of large enterprises are beginning to incorporate machine learning models across a number of products. These models are typically trained on shared, multi-tenant GPU clusters. Similar to existing cluster computing workloads, scheduling frameworks aim to provide features like high efficiency, resource isolation, fair sharing across users, etc. However Deep Neural Network (DNN) based workloads, predominantly trained on GPUs, differ in two significant ways from traditional big data analytics workloads. First, from a cluster utilization perspective, GPUs represent a monolithic resource that cannot be shared at a fine granularity across users. Second, from a workload perspective, deep learning frameworks require gang scheduling reducing the flexibility of scheduling and making the jobs themselves inelastic to failures at runtime. In this paper we present a detailed workload characterization of a two-month long trace from a multi-tenant GPU cluster in Microsoft. By correlating scheduler logs with logs from individual jobs, we study three distinct issues that affect cluster utilization for DNN training workloads on multi-tenant clusters: (1) the effect of gang scheduling and locality constraints on queuing, (2) the effect of locality on GPU utilization, and (3) failures during training. Based on our experience running a large-scale operation, we provide design guidelines pertaining to next-generation cluster schedulers for DNN training workloads.

1 Introduction

Recent advances in machine learning have led to tremendous improvements in tasks ranging from object detection [31] to speech recognition [34] and language translation [47]. As a result a number of enterprises are now incorporating machine learning models in various products [1,4]. To facilitate model training, enterprises typically setup a large cluster shared by users belonging to a number of different production groups. Similar to clusters setup for big data analysis [12,50], using shared clusters can facilitate better utilization and reduce development overheads.

However deep learning workloads pose a number of new requirements or constraints on cluster management systems. Since machine learning algorithms are floating point computation intensive, these workloads require hardware accelerators like GPUs. However, unlike CPUs, accelerators do not typically have proper hardware support for fine-grained sharing [21]. While there are software mechanisms to enable sharing, they often have high overhead making it challenging to share resources across jobs [40,53]. Furthermore, training on large datasets often requires the use of multiple GPUs [20] and machine learning frameworks typically require that tasks on each GPU be scheduled at the same time, i.e., gang scheduled [18]. This increases the risk of resource fragmentation and low utilization in shared clusters. Finally, multi-GPU training also implies synchronization of model parameters across GPUs and hence it is important to achieve better *locality* while scheduling to allow for the use of faster interconnects for both intra- and inter-machine communication.

Despite their growing popularity, to the best of our knowledge, there has been no systematic study of multi-tenant clusters used to train machine learning models. In this paper, we present the design of a large, multi-tenant GPU-based cluster used for training deep learning models in production. We describe Philly, a service in Microsoft for training machine learning models that performs resource scheduling and cluster management for jobs running on the cluster. Using data from this system, we then present a detailed workload characterization and study how factors such as gang scheduling, locality requirements and failures affect *cluster utilization*.

Our analysis spans across two months and uses around 100,000 jobs run by hundreds of users. We combine logs from Apache YARN [48], our cluster scheduler, utilization information from Ganglia [33], and logs from each job to perform a systematic analysis of cluster utilization.

We study two main aspects of how locality-aware scheduling affects performance and utilization. First, we study how waiting for locality constraints can influence queuing delays

before training jobs are run. Training jobs need to be gang scheduled, as hyper-parameters are picked for specific GPU count configurations. Given that training jobs take a long time to run, and greater locality improves performance due to the availability of faster interconnects for parallel training [52], the scheduler in Philly waits for appropriate availability of GPUs before beginning to run the training job. Our study shows that as one might expect, relaxing locality constraints reduces queueing delays, especially for jobs that use many GPUs – our emphasis here is not on presenting this as a new insight, but instead on highlighting this using real-world data from production clusters.

Next, we study how locality-aware scheduling can affect the GPU utilization for distributed training jobs. Even though most GPUs within a cluster are allocated to users, thus *suggesting* high cluster utilization, this metric alone is misleading. We show that the hardware utilization of GPUs in use is only around 52% on average. We investigate two reasons which contribute to low GPU utilization: (1) the *distribution* of *individual* jobs across servers, ignoring locality constraints, increases synchronization overheads, and (2) the *colocation* or packing of *different* jobs on same server leads to interference due to contention for shared resources.

Finally, we look at why jobs might fail to complete successfully and offer a detailed characterization of the causes for such failures in our clusters. Around 30% of jobs are killed or finish unsuccessfully due to failures. Failures are caused by errors across the stack, with programming errors dominating failures and occurring early in the training process; failures due to cluster components like HDFS tend to occur much later in the training lifecycle.

Based on the lessons learnt from data analysis and our experiences running a large-scale operation over the years, we provide three guidelines to improve the next generation of cluster schedulers for DNN workloads. First, because the lack of locality impacts both utilization and job runtime, and because DNN training jobs are long running, schedulers should trade queueing delay for adhering to locality constraints. Second, different jobs that share a single server may interfere with each other and thus adversely affect their training time. Schedulers should thus aim to isolate the jobs on dedicated servers while implementing techniques like migration for defragmentation, to support the locality constraints of jobs that need more GPUs. Third, many failures ought to be caught early, well before they are scheduled on a larger shared cluster. This can be achieved by scheduling each incoming job on a small dedicated pool of servers or even using a single GPU should be able to catch simple programming and configuration errors from multi-GPU jobs. Furthermore, an online analysis of failures at runtime can let schedulers adapt their retry policies thus avoiding wasteful re-execution.

Philly's design does not stand in isolation. There are many open platforms for DNN job scheduling that use designs similar to Philly, e.g., OpenPAI [36] and Submarine [44]. We hope that insights and data from our study, and the accompanying traces, inform the burgeoning work of scheduling research for machine learning workloads.

2 Philly: System Overview

In this section we provide an overview of the design and architecture of Philly. First, we describe the workloads that are supported in our system and then describe the hardware characteristics of the clusters. Next, we describe the lifecycle of a job. Finally, we explain our data collection pipeline and highlight the data we use to perform our analysis in subsequent sections. The authors would like to note that Philly has been developed over the past few years by a team of developers in our company and has gone through multiple generations of design.

2.1 Workloads

Philly is designed to support workloads that perform supervised machine learning where jobs learn a model given training data and the corresponding labels. This includes training jobs from production groups developing products that use models for image classification, speech recognition, etc. The system supports jobs written using any machine learning framework like TensorFlow [5], CNTK [42], Caffe [29], and PyTorch [39]. Jobs are based on recently proposed learning architectures like convolutional neural networks [31], LSTMs [45] and RNNs [35].

All jobs, irrespective of the framework or model being used, rely on iterative optimization methods [19] like stochastic gradient descent (SGD). In each iteration, the gradient computation is performed by translating the model components into code that can be executed on accelerators like GPUs. The gradient values are then aggregated to compute a model update and these iterations are repeated until convergence. Training a model could require thousands to millions of iterations [46], and result in multiple passes or *epochs* over the entire dataset.

To scale training across larger datasets, a number of jobs use distributed training across machines. Distributed training typically uses data parallelism where each worker loads a complete copy of the model into its own memory. In each iteration, every worker performs training using a subset of the input data, and at the end of the iteration all workers exchange gradients to synchronize model updates. This synchronization phase is performed using either parameter servers [32] or high performance libraries for collective communication (such as MPI, NCCL, etc).

2.2 Cluster Architecture

Our system is deployed on large GPU clusters shared across many groups in the company. Our clusters has grown significantly over time, both in terms of the number of machines

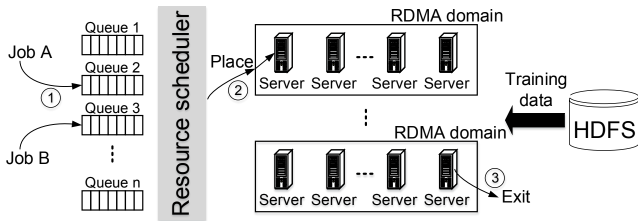


Figure 1: The lifecycle of deep learning jobs in Philly.

(5× increase in one year) as well as the number of GPUs per machine (2-GPU to 8-GPU servers).

Our clusters have high-speed network connectivity among servers and GPUs in the cluster. This is to speed up distributed training where workers need to exchange model updates promptly for every iteration. There is a hierarchy of network links available in our cluster for communication across GPUs. For example, machines within the same rack (*RDMA domain*) are connected via 100-Gbps RDMA (InfiniBand) network, while cross-rack traffic goes through Ethernet. To improve communication performance, workers in a distributed training job must either be colocated on the same machine or preferably communicate over a higher-speed network such as say InfiniBand. Thus, our framework considers both GPUs and network connectivity for scheduling.

Similar to existing big data analytics clusters, our clusters use HDFS [43] as the distributed storage system and our resource manager is based off Apache YARN [48]. Input data for the machine learning jobs is stored in HDFS and read by jobs during training. Users provide a Docker container with their training code and its dependencies. Each training job requests 1 or more GPUs which can be allocated across multiple machines. Philly instantiates one container per machine allocated to the job when it is scheduled for execution.

2.3 Job Scheduling and Execution Workflow

Figure 1 shows the lifecycle of a deep learning job in Philly and the various stages of execution that it goes through.

Incoming jobs and queueing ①. As a part of job submission, users specify *the number of GPUs* required. To facilitate host resource allocation, we perform an allocation of CPU cores and memory capacity proportional to the requested GPU count. Once a job has been received by the scheduler it is queued while the necessary GPUs are allocated. To support multiple production groups we create a virtual cluster for each group and associate a *resource share* or *quota* in terms of number of GPUs to each virtual cluster. Each virtual cluster has a separate allocation queue in Apache YARN and we use the Fair Scheduler to manage these queues [2]. Our scheduler not only respects the configured resource shares but also allocates unused GPUs to a queue which has additional demand. Jobs can be preempted based on fair share of resources among virtual clusters. Our scheduler starts preemption only when a majority (90%) of total GPUs are being used.

For distributed learning, deep learning frameworks require all the GPUs to be available at the same time [22]. Thus the scheduler needs to perform *gang scheduling* while being *locality-aware*, i.e., pack a job's GPUs onto the smallest number of servers and within an RDMA domain. Locality awareness improves training time by bringing down the time needed for parameter synchronization [22,52] due to the availability of: (i) fast intra-server interconnects (such as PCIe and NVLink), and (ii) for jobs that do not fit on a single server, high-bandwidth links available within an RDMA domain. We implement these goals by acquiring resources for a job as GPUs become available and waiting for a pre-specified timeout (2–3 minutes in our setup) to acquire all the necessary GPUs with the locality constraints. To facilitate locality-aware GPU scheduling, our job scheduler keeps track of all idle GPUs in the cluster and ranks the corresponding racks and servers. Specifically, racks are ranked by increasing order of allocation or occupancy, and the machines in a rack are ordered the same way. This allows the scheduler to first consider racks and then servers within those racks that have most GPUs available.

If the request is not fulfilled by the timeout, any partially acquired resources are relinquished and we retry scheduling after a back-off (2 minutes in our setup). To avoid starvation, the locality constraints are relaxed after a scheduling request has been retried a fixed number of times. We analyze corresponding queuing delays in Section 3.1.

Job placement and utilization ②. While the scheduler tries to maximize locality for distributed jobs as described before, at the same time the scheduler also aims to avoid fragmentation of resources from smaller jobs (e.g., 1-GPU jobs) by packing them into a fewer servers. However colocating different jobs on the same server could lead to lower GPU utilization due to interference in shared system resources such as PCIe bus [52]. In order to better understand this trade-off we study the effects of colocation vs. distribution and measure how that affects utilization.

Once the job is scheduled to run, its GPUs are *not shared* with other jobs. This is because model training can be computation intensive and we need consistent performance among workers of the job without having stragglers. However, dedicated GPUs may be underutilized for many reasons, e.g., inefficiencies in the code generated by the machine learning frameworks or programs blocking on I/O when reading data from storage. GPU underutilization also comes from distributed training where computation may block during model synchronization among the workers. We analyze the effects of job placement and GPU utilization in Section 3.2.

Table 1 qualitatively compares Philly with the state-of-the-art DNN cluster schedulers, showing both similarities and differences exist. Nonetheless, locality and colocation are the common issue for all contemporary clusters, and that insights

Table 1: Comparison of DNN cluster schedulers. JCT means job completion time.

	Philly	Gandiva [52]	Optimus [38]	Tiresias [22]
Objective	Consolidation	Consolidation	Average JCT	Average JCT
Algorithm	Locality-based	Time-sharing	SRTF	Gittins Index & LAS
Input	Arrival time	N/A	Remaining time	Attained service
Preemption	Model checkpoint	Context switch	Model checkpoint	Model checkpoint

obtained in this study are widely valuable.

Training progress and completion ③. Jobs can finish with one of three statuses: passed, killed, or unsuccessful. Passed indicates that the job completed successfully, while killed indicates that the job was terminated by the user.

Among successful jobs, every job runs a number of iterations to improve the model incrementally, and the number of iterations to run is typically a static parameter set by the user. In cases where a job is configured with too many iterations, it is possible to deliver the same (or similar) quality of trained model with fewer iterations. Failed jobs in our system are retried a fixed number of times. This is useful for overcoming non-deterministic failures and if the job does not succeed after retries then it is marked as unsuccessful. As failures also contribute to ineffective cluster utilization, we perform a detailed study to understand the reasons behind failures in Section 4.2.

While our focus in this section is specifically about the lifecycle and execution flow in Philly, there are many open platforms for ML job scheduling that use a similar design. Platforms like OpenPAI [36] and Submarine [44] also use a centralized scheduler with support for running machine learning frameworks as Docker containers. While the details of the scheduling algorithm vary across systems, a number of aspects we study in this paper are independent of the choice of scheduler: e.g., failures due to programming errors and bugs in popular frameworks, effect of distributed training across machines, etc. Thus, we believe that lessons from Philly are generally applicable to other clusters as well.

2.4 Data Collection and Analysis

The cluster under study consists of hundreds of machines accounting for thousands of GPUs of the same model. The cluster has 2 server SKUs – one with 2 GPUs per server and another with 8 GPUs per server; RDMA domains are homogeneous with respect to server SKUs. To get a comprehensive understanding of the characteristics of our system and workloads, we developed a data collection and analysis pipeline and collect logs over a 75-day period from Oct. 2017 to Dec. 2017. Our logs contain a total of 96260 jobs over 14 virtual clusters.

The analysis pipeline combines three main log sources in our system as follows. (1) We collect the YARN scheduler logs to obtain job arrival time, number of GPUs requested, GPU allocation status, and job finish status. (2) We collect stdout and stderr logs from the machine learning frameworks

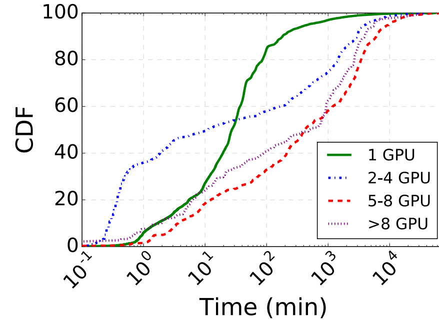


Figure 2: CDF of job run times for 1 GPU, 2-4 GPU, 5-8 GPU, and >8 GPU jobs.

that execute scheduled jobs. (3) We collect logs from Ganglia monitoring system that reports per-minute statistics on hardware usage on every server, including CPU, memory, network, GPU utilizations. Combined with GPU allocation status in YARN scheduler logs, we can track how a scheduled job utilizes cluster hardware resources.

Our collected data contains jobs from a wide spectrum in terms of their run times and sizes, and consequently cluster resources demands. Jobs run from minutes to days or even weeks, as shown in Figure 2. In contrast, in big data analytics, job execution times range from only tens of milliseconds to a few hours [11, 37, 41]. Furthermore, we see that our workload has significant skewness in run time, with 0.5% jobs taking more than a week to be finished. Figure 2 also shows how jobs of different sizes vary in terms of execution times. We see that jobs with more GPUs tend to run longer. This results in most of the cluster resources demands coming from the larger jobs, and resource availability status changing relatively slowly over time.

3 Impact of Locality Awareness

Our scheduler trades off locality for lower waiting. Thus placement choices made by the scheduler affects the efficiency of DNN training in two parts: queueing delay (before job execution) and hardware utilization of in-use GPUs (after job execution). The effect of locality constraints on queuing delays has been extensively explored in large-scale resource allocation [7, 11, 26, 54]. Machine learning workloads introduce similar constraints driven by gang scheduling and the requirement for using fast interconnects. In Section 3.1, we an-

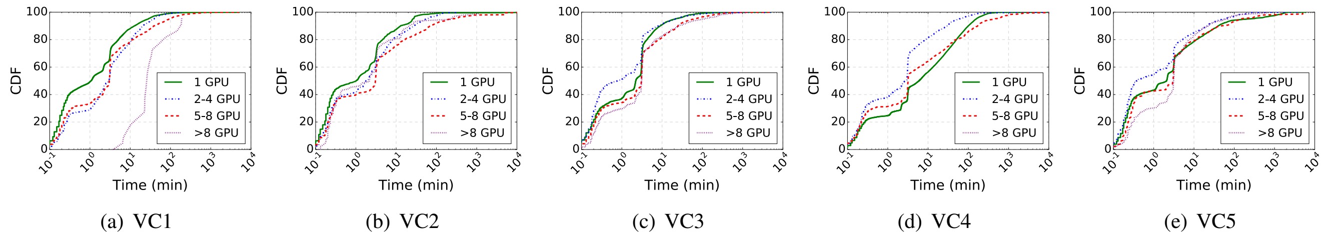


Figure 3: CDF of scheduler queueing delay for five of the largest virtual clusters in our deployment. Note that VC4 contains no jobs with >8 GPU.

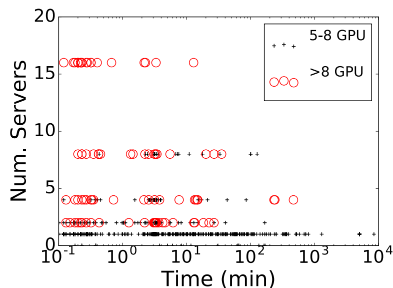


Figure 4: For a given GPU count, relaxing locality constraints reduces queueing delays (VC2).

Delay	2-4 GPU	5-8 GPU	>8 GPU
Fair-share	5168 (40.6%)	3793 (25.8%)	66 (2.1%)
Fragmentation	7567 (59.4%)	10928 (74.2%)	3117 (97.9%)

Table 2: Frequencies of two types of queueing delay.

alyze queueing delays in the context of DNN training cluster using real-world data in detail. Next, we study utilization of processing cycles for GPUs allocated to training jobs in Section 3.2. In particular, while prior work discusses efficiency of distributed training for a certain job size or a configured placement [22, 52], we perform an analysis on the aggregated efficiency for a range of job sizes for the first time.

3.1 Queueing Delays

We first consider overall queueing delay observed during job scheduling. We plot the CDF of queueing delay in Figure 3 for all jobs in five of the largest virtual clusters (VCs). Jobs that need more than 4 GPUs tend to have a slightly longer tail in the distribution of queueing delays compared to their 1 GPU and 2-4 GPU counterparts. For example for VC2, 25% of jobs using >4 GPUs, which include both 5-8 GPU and >8 GPU, experience a queueing delay of at least 10 minutes; in comparison, only 10% of 1 GPU jobs experience a queueing delay of at least 10 minutes.

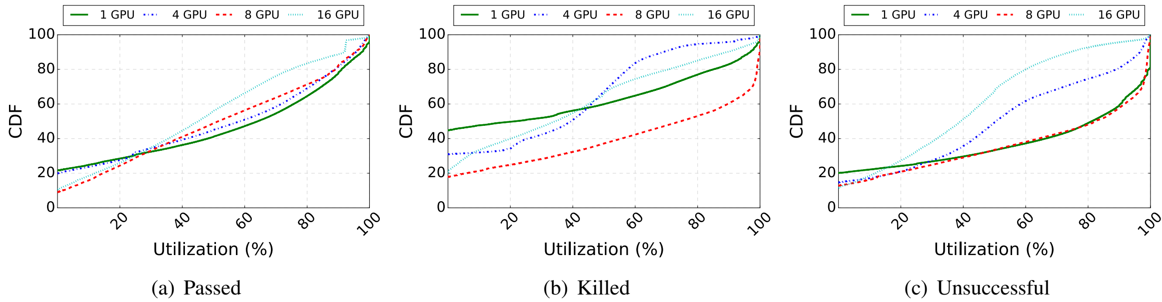
But overall, queuing delays for jobs, irrespective of their GPU demand, are not markedly distinct. This is partially a consequence of our scheduling policy that chooses to relax locality constraints in order to start a job without incurring a very long queueing delay penalty. To highlight the relation between locality constraints and queueing delays, we next consider jobs with 5-8 GPU and >8 GPU. We correlate scheduler waiting times with number of servers on which the jobs are placed, and show the results in Figure 4. As expected, most of jobs with 5-8 GPU are scheduled with high locality, i.e., placed on one or two servers. On the other hand, we find that jobs with >8 GPU are spread across a wider range from 2 to 16 servers. Clearly, when jobs end up running on 16 servers, they start execution much sooner than running on 2 or 4 servers. This confirms how our scheduler works in practice to trade-off locality for lower scheduling delay.

While effective, we find that this decision affects the GPU utilization as discussed in Section 3.2. We next look at more details on the queuing delay characteristics and break down the delay by different causes.

3.1.1 Impact of Locality-Driven Scheduling

Queuing delay can be caused by two primary factors: fairness (which is common in conventional data analytics clusters), and locality requirement and resource fragmentation (which is more prevalent in deep learning clusters). We call queueing caused by the first factor as *fair-share delay*, as it happens when the virtual cluster uses up its assigned quota (i.e., number of GPUs). However, it is possible that a job arrives within the quota but fails to be scheduled, mainly because resource fragmentation makes it hard to find enough GPUs with high locality. We call this queuing delay as *fragmentation delay*. In practice, we find that resource fragmentation is very common. For example, we observe that (i) when two thirds of the total GPUs are being used, the fraction of servers that are completely empty is less than 4.5% and that (ii) these servers are spread across RDMA domains.

We next see how frequently fair-share delay and fragmentation delay occur for different job sizes in our workloads. Since some jobs are quickly terminated, we only consider jobs that run for at least one minute. Further, since fragmentation influences distributed training jobs only, we consider jobs that



(a) Passed (b) Killed (c) Unsuccessful

Figure 5: CDF of per-minute GPU utilization for passed, killed, unsuccessful jobs in different sizes.

use 2 or more GPUs. Table 2 shows the frequencies for the two types of delay. For jobs with 5-8 GPU, fragmentation delay is responsible for 74.2% of occurrences, and it dominates for larger jobs. In contrast, for smaller jobs, we see that the two causes are more balanced. Further, we also observe that across all jobs fragmentation delay is responsible for around 80% of the delay in terms of waiting time. This is because fair-share delays are easy to mitigate with preemption, but fragmentation delays are much harder to overcome in our current design.

Finally, we note that the queuing delay fractions vary across virtual clusters. Among the five largest virtual clusters, VC5 often over-subscribes its quota and thus the proportion of fair-share delay is overall higher at 37%.

Does out-of-order scheduling exacerbate job queueing? Given the resource fragmentation and the fact that the YARN scheduler is work-conserving, larger jobs could be additionally negatively affected by out-of-order scheduling. To see how, consider a job that requires 24 GPUs spread across three machines. While this job is waiting for such configuration, if a smaller job requests 2 GPUs, it is scheduled on machines where two GPUs become available. This could cause further fragmentation and lead to the 24-GPU job needing to retry after a backoff. In our workload, out-of-order scheduling is quite common, with 38.1% of scheduling decisions, and occurs 100% for jobs with 5-8 GPU or >8 GPU. However, we find that most out-of-order scheduling decisions do not greatly affect the waiting time for resource-intensive jobs. For example, for out-of-order scheduling occurrences of jobs with 5-8 GPU or >8 GPU, as much as 85.0% corresponds to cases where idle GPUs are effectively utilized without prolonging the scheduling time of those waiting jobs.

In summary, our analysis shows why it makes sense to relax locality over time to mitigate queuing delays for distributed training. We also find that in addition to fair-share queuing delay, the need for gang scheduling and locality introduces fragmentation delay for machine learning jobs.

Job size	Passed	Killed	Unsuccessful	All
1 GPU	53.51	37.02	62.82	52.38
4 GPU	51.13	34.39	50.95	45.18
8 GPU	51.09	60.63	64.34	58.99
16 GPU	44.88	36.98	39.02	40.39
All	52.43	42.98	60.43	52.32

Table 3: Mean GPU utilization for different job sizes.

3.2 GPU utilization

GPUs are the most expensive resources in our cluster and this makes their efficiency an important factor in assessing the cost-effectiveness across the entire cluster. For each individual GPU, Ganglia [33] reports aggregate performance counters every minute, including utilization of processing cycles and memory, temperature, power usage, etc [3]. We next present how efficiently training jobs use processing cycles in their (exclusively) allocated GPUs. Note that our current generation of GPUs only report coarse-grained utilization for processing cycles that can only be used to detect if any of the streaming multiprocessors (SMs) are being utilized [3]. They do not report what fraction of the SMs are being actually used within a single GPU. Therefore, our analysis presents an "upper bound" of actual effective SM utilization.

Overall, deep learning training jobs underutilize GPU processing cycles regardless of their job sizes. Figure 5 shows CDFs of per-minute GPU utilization of passed, killed, and unsuccessful jobs for different sizes. Table 3 reports averages for each job size, including averages for different job status; we use these job sizes as representative of small, medium and large jobs based on the GPU request distribution in our cluster. Surprisingly we find that around 47.7% of in-use GPUs' cycles are wasted across all jobs, with jobs using 16 GPUs exhibiting the lowest utilization at 40.39%. Moreover, across job status in Figure 5, the median utilization for 16 GPU jobs is 45.00%, 34.24%, 39.54% for `Passed`, `Killed`, and `Unsuccessful`, respectively. These are 6.46%, 40.25%, and 42.63% lower than the 8 GPU jobs in the corresponding job status. We study the efficiency of such jobs in the next section in detail.

Metric	SameServer	DiffServer	IntraServer	InterServer
GPU util.	57.7	49.6	37.5	36.5
Images/s	114.8	98.0	75.6	74.1

Table 4: Mean GPU utilization and training performance of ResNet-50 over different locality/colocation configurations.

3.2.1 Impact of Distributed Learning

Given that the 8 GPUs mounted in each server can communicate more efficiently without using the network, our job scheduling strategy is to favor intra-server locality when assigning each job to available GPUs. At the same time, the scheduler attempts to pack small jobs into fewer servers to avoid fragmentation. This leads to *job colocation* on the same server and consequently could lead to interference in shared system resources (e.g., RDMA and PCIe) [52]. This creates an interesting utilization spectrum for multi-GPU jobs. In particular, jobs using more than 8 GPUs must *distribute* training instances across multiple servers and may be dynamically colocated with other jobs. This scenario also involves communication overheads since each server has to periodically wait for model aggregation to happen over the network.

To confirm that such distribution and colocation factors indeed relate to the efficiency of GPUs in use, we first characterize utilization of processing cycles for various job placement scenarios using a popular image recognition model, ResNet-50 [23]. Specifically we train ResNet-50 with 2 GPUs using TensorFlow and perform offline experiments with placements that exercise shared resources differently. Then using our telemetry data, we attempt to infer correlations between those factors and the observed efficiency in our cluster.

Analysis using ResNet-50. Table 4 shows the impact of distribution only, by comparing a ResNet-50 job placed in a single server (`SameServer`) with the job placed in two servers connected with RDMA network (`DiffServer`). Each server has four NVIDIA Tesla P100 GPUs attached to a CPU socket. The table reports GPU utilization when processing a batch size of 32 images during training. First we observe that the training does not fully utilize GPUs even for single machine execution. In particular, `SameServer` achieves utilization of 57.7% for GPUs in use. It increases to 71.1% for twice the batch size but only increases marginally for larger batches. Also the table shows that using distributed training achieves lower utilization of 49.6% in `DiffServer`. This shows that even for 2-GPU jobs, there is a cost to not achieving locality.

Given a distributed training setup, contention for shared resources like RDMA and PCIe further lowers the efficiency of utilized GPUs. To show this we set `DiffServer` as our baseline and measure changes in the efficiency while populating additional ResNet-50 jobs in the same servers. First, we measure GPU utilization when the colocated jobs do not use RDMA network at all: we place two `SameServer` jobs, one on each server in the same CPU socket as the job under

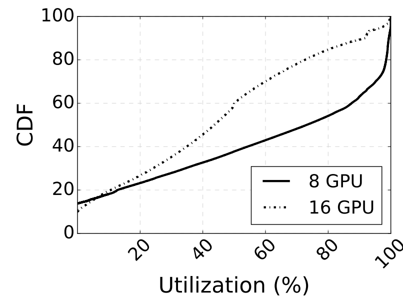


Figure 6: GPU utilization when running 8 and 16 GPU jobs on dedicated servers.

study. Thus, these jobs interfere with the job under study in the use of PCIe buses while reading training inputs, aggregating model updates, and so on. The observed efficiency is shown as `IntraServer` in Table 4, and we see that having such intra-server interference lowers the utilization by as much as 12.1%. We also study if such interference matters for the RDMA network in `InterServer`. For this setup we use two `DiffServer` jobs instead of two `SameServer` jobs as background traffic, so that all the jobs are distributed across two servers and share the RDMA network. In this case, we see a 13.1% decrease in utilization compared to the baseline.

Our experimental study reveals that efficiency of allocated GPUs varies according to locality and colocation scenarios that could occur in the cluster. Further, any placement that causes lowered GPU utilization also results in slowdown in training performance (i.e., images processed per second) as shown in Table 4. Next, we analyze utilization for our aggregate workload. We note that unlike the controlled experiment, the type of model trained and the batch sizes used vary across jobs in our aggregate workload making it harder to establish a baseline utilization without distribution or inference.

Distributed training with dedicated servers. First, to study the effects of distribution, we restrict our study to look at 8 GPU and 16 GPU jobs that are packed on one or two servers. In this case, the 8 GPU jobs uses all 8 GPUs in a single server while the 16 GPU jobs uses all the GPUs in two servers. The network over which the servers for these jobs are connected to each other is shared. Figure 6 shows the results of our comparison. Compared to the 8 GPU jobs, we see that 16 GPU jobs, which have the additional model aggregation step in distributed mode, have significantly lower utilization. Specifically, for 8 GPU jobs, GPU cycles are utilized 56.9% of time on average while this is only 34.3% for 16 GPU jobs. Furthermore, the median is 73.12% for 8 GPU jobs, which is 1.67x the median in the 16 GPU case.

Distributed training with shared servers. When locality constraints are relaxed, a job may have to be distributed over many servers while sharing them with other jobs. Distributing a job over many shared servers can further lower utilization of GPUs. This drop in utilization occurs not only due to a higher network overhead but also because of interference from

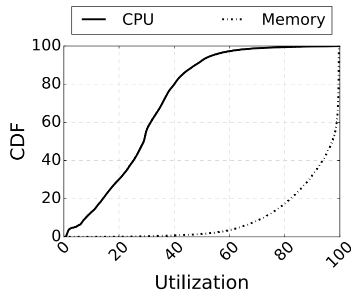


Figure 7: Host resource utilization.

Degree	Mean	50%ile	90%ile	95%ile
2 servers	43.66	43.69	91.77	97.06
4 servers	40.94	39.85	83.28	91.97
8 servers	28.56	25.71	65.68	78.85

Table 5: GPU utilization for 16-GPU jobs that are spread over 2, 4, and 8 servers.

unrelated but co-located jobs. To study this, we see how the GPU utilization of 16-GPU jobs varies as as we move from dedicated GPUs to a larger number of shared servers. Table 5 shows the average and percentiles for GPU utilization across the different allocation scenarios.

When running on 2 8-GPU servers, a 16-GPU job has dedicated servers. When running on 4 servers, the 16-GPU job may occupy 4 GPUs on each server, and will be colocated with other jobs on those servers. We find that the degree of interference is larger if the job is distributed on more servers. Table 5 shows that in addition to the inefficiency caused by distribution (Figure 6) there is additional underutilization caused by colocation. We see that for 16-GPU jobs distributed across 8 servers, the average utilization is as low as 28.26% and more than 90% of jobs have less than 66% utilization.

Among host resources, our scheduler dedicates CPU and memory along with GPU to each job. In deep learning clusters, these host resources are used for many useful tasks including caching training inputs, model aggregation, and periodic model validation and progress report. By default, we allocate CPU and memory capacity proportional to the number of requested GPUs. Figure 7 shows CDFs of utilization of these host resources observed in our servers. In general, many servers underutilize CPU cycles yet highly utilize memory. This indicates that a useful feature in the scheduler would be to observe if a particular job requires disproportionate amount of host memory and isolate memory used by jobs colocated on the same server.

In summary, our data analysis shows how GPUs are underutilized in shared clusters. We presented correlations of how distribution and interference affect utilization and validated this using a controlled experiment to break down the importance of locality and interference. We discuss some implications for scheduler design in Section 5.

Status	Count(%)	GPU times used (%)
Passed	66696 (69.3%)	44.53%
Killed	12996 (13.5%)	37.69%
Unsuccessful	16568 (17.2%)	17.76%
Total	96260 (100.0%)	100.0%

Table 6: Distribution of jobs by their final status.

4 Training Progress and Completion

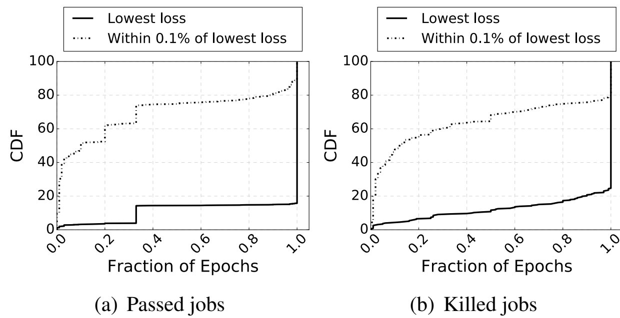
Jobs in our system finish with one of three statuses: passed, killed or unsuccessful. Similar to iterative online computations [6, 16], our machine learning job utilizes cluster resources to improve the model over time. However as opposed to prior study on big data traces [30], we see a significant fraction of jobs (30.7% as shown in Table 6) are either terminated unsuccessfully or killed by users. They constitute around 55% of the total GPU time used during our trace collection period. Thus it is important to understand the reason behind these failures as fewer unsuccessful jobs would mean that more of the cluster resources can be used for successful jobs.

4.1 Effectiveness of Training Iterations

Most deep learning jobs optimize a non-convex loss function and the optimization algorithms do not necessarily guarantee that the loss always decreases with more training. Thus, similar to [22], users in our system submit model training jobs using a larger number of epochs than necessary to get the optimal model. To analyze the magnitude of this effect we study how the training loss for a job varies across epochs and measure the epoch at which we achieve the best training loss. As this information is not printed in the log by every user/framework, we are only able to obtain convergence information for around 2502 jobs.

First, Figure 8(a) shows the fractions of epochs required to reach the lowest loss across all passed jobs. From the figure we see that around 80% of passed jobs require all the epochs executed to reach the lowest loss. We repeat this study for killed jobs and see a similar pattern as shown in Figure 8(b).

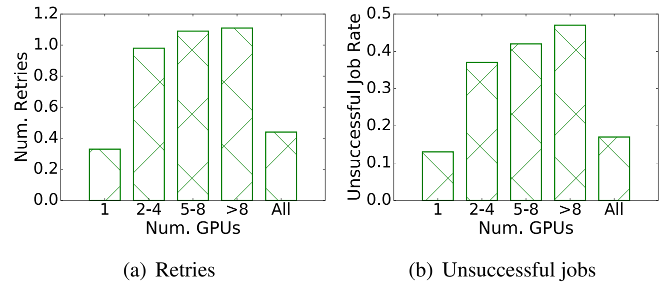
However we also see that a majority of jobs improve the loss marginally using a large fraction of epochs. In particular, Figure 8(a) shows the fraction of epochs required to reach within 0.1% of the lowest loss across all passed jobs. Around 75% of jobs reach within 0.1% of the lowest loss using only 40% of the epochs. Again, a similar pattern is shown for killed jobs in Figure 8(b). While we do not present data from user surveys, this suggests that machine learning practitioners can early terminate jobs to save use of GPU times considerably when the loss change is less than a particular threshold in successive epochs. Essentially, we look into much resources are used to improve 0.1% of convergence accuracy in terms of the fraction of GPU times for each job. In our workload, this accounts for 62% and 56% on average for passed jobs and killed jobs, respectively.



(a) Passed jobs

(b) Killed jobs

Figure 8: Fraction of epochs necessary to achieve a particular loss threshold for (a) passed jobs and (b) killed jobs.



(a) Retries

(b) Unsuccessful jobs

Figure 9: (a) Average number of job retries for using different number of GPUs, and (b) subsequent unsuccessful jobs.

4.2 Job Failures

We next present a detailed analysis on job failures, including why/when/how frequently jobs fail and what their impact is on effective cluster usage. We remind the reader that in our cluster scheduler, a job is retried upon failure. If the job repeatedly fails it is marked as unsuccessful as further retries are deemed no longer effective. Figure 9 presents a high-level summary of job retries/failures and shows that jobs using more than 4 GPUs not only retry execution more often but also finish unsuccessfully at higher rate. The reasons behind job retries/failures are diverse, and failures occur at different times during job execution. We thus investigate failures by classifying them across layers of our system stack.

4.2.1 Failure Classification

Table 7 presents analysis results of failures based on two classification factors. First, failures are classified from different sources (Column 2): the sources include (i) Infrastructure (`IF`) which includes YARN, HDFS and all other framework components, (ii) AI Engine (`AE`) which includes TensorFlow, Torch, and any other platforms, and (iii) User (`U`) which represents programmers. Column 1 lists a number of reasons for failures we observe from the workload.

Most failure reasons in the table are self-explanatory, and we describe six important ones in more detail here.

(1) **Incorrect inputs**: Model files or input data stored in the external HDFS storage cannot be read.

(2) **Semantic error:** Errors that happen due to library version mismatch or other dependencies of the user training program not being setup correctly.

(3) **Model checkpoint error:** The job is not able to successfully create a model checkpoint after a certain number of epochs complete. This is usually due to either transient error in HDFS or HDFS name node recovery.

(4) **MPI runtime failure:** This is usually due to either a failure of network connection to peer MPI process, or possibly an internal failure of the MPI daemon itself.

(5) **Job preempted:** YARN reclaims any GPU currently in use to schedule another job.

(6) **Invalid memory access:** Training job dies because of violating access on memory address space e.g., using an invalid pointer value, or having race condition while copying data. This failure is observed in both CPU memory and memory allocated for GPU access.

While bridging failure category and failure reason, we observe that a failure reason can appear in multiple categories, even in all involved categories, as shown in Column 2 of Table 7.

Building failure classifier. There exists causality among various failure reasons. For example, *traceback from crash* is a consequence of an *invalid memory access*. Our first mission in building a classifier is identifying signatures of failure reasons closer to the root cause. We capture root-cause signatures from stdout or stderr logs of a failed job. If not explicit from the logs, we then attempt to capture implicit ones such as *traceback from crash*. In consequence, our classifier has in total **more than 230 rules** to find both explicit signatures and implicit signatures. If there is no signature for a failure, we tag it as *no signature*, which constitutes 4.2% of the total failures.

4.2.2 Failure Frequency

Column 3 of Table 7 summarizes the occurrence frequency of the classified failure reason. `Trial` counts the number of failure events observed in our workload: failure reasons are sorted by it. We further group `Trial` occurrences by job ID (`Job`) and user ID (`User`) to see if failures are localized according to the same job or user.

Failures repeat for the same job/user. Our analysis shows that across failure reasons, failures repeat at both job level and user level. In particular, we measure repetition factors (i.e., `Trial` divided by `Job` or `User`) for top-8 failure reasons, which cover 88.9% of the total failures. The measured repetition factors are 2.3 and 38.8 on average for `Job` and `User`, respectively, meaning a single job and a single user on average cause 2.3 and 38.8 occurrences of failure, respectively, during

Failure Reason	Category			Num Occurrences			RTF: Runtime to Failure (mins)				GPU Demand			RTF×Demand (%)
	IF	AE	U	Trial	Job	User	50%ile	90%ile	95%ile	Total %	1	2-4	>4	
CPU out of memory		✓	✓	12076	2803	65	13.45	17.73	33.97	6.62	11465	235	376	3982320 (8.05)
Incorrect inputs	✓		✓	9690	4936	208	1.87	404.83	2095.73	30.43	5844	2638	1208	11979474 (24.21)
Semantic error	✓		✓	2943	2049	159	2.72	376.00	1436.88	9.22	1603	494	846	8442835 (17.06)
Core dump		✓	✓	2912	1784	122	0.85	72.75	431.65	3.35	1936	496	480	1493632 (3.02)
Invalid mem access			✓	2602	1235	108	1.03	403.50	1357.38	3.82	712	774	1116	2352994 (4.75)
Model ckpt error	✓			1995	948	85	181.67	3728.93	8196.02	21.73	743	384	868	8080374 (16.33)
CUDA failure		✓		1484	571	70	1.32	19.87	82.17	0.62	133	1153	198	357119 (0.72)
Syntax error	✓		✓	1132	883	110	0.58	5.02	12.00	0.19	780	184	168	130094 (0.26)
Traceback from crash	✓	✓	✓	777	271	44	1.02	894.33	1394.07	2.34	356	277	144	863130 (1.74)
MPI error	✓			634	166	28	1.62	3015.27	5143.98	3.70	456	54	124	613059 (1.24)
GPU out of memory		✓		487	261	35	18.53	353.62	2740.28	1.08	237	70	180	1040249 (2.10)
MPI runtime failure	✓			478	420	96	1389.48	13778.60	18090.88	14.63	240	141	97	7593398 (15.34)
Permission error			✓	299	151	37	1.00	8.15	15.85	0.07	56	202	41	15185 (0.03)
Import error	✓		✓	148	148	41	0.67	4.58	10.73	0.06	108	30	10	10803 (0.02)
Job preempted	✓			147	95	34	559.08	2682.85	5892.23	1.66	25	95	27	2338772 (4.73)
CUDA init failed		✓		141	69	20	1.08	2.18	4.63	0.03	16	66	59	64512 (0.13)
Model diverged			✓	84	30	5	1.48	44.37	76.53	0.01	78	5	1	2562 (0.01)
CUDA ver. mismatch		✓		49	49	19	0.83	1.65	1.67	0.00	1	1	47	421 (0.00)
GPU ECC error		✓		10	10	2	26.82	671.92	2035.02	0.03	1	5	4	23575 (0.05)
Output node error			✓	3	3	1	0.85	0.95	0.95	0.00	3	0	0	2 (0.00)
Cannot load libs		✓		1	1	1	0.12	0.12	0.12	0.00	1	0	0	0.12 (0.00)
No signature				1684	698	94	1.87	28.00	95.17	0.42	1235	294	155	102138.03 (0.21)

Table 7: Failures classified into failure reasons (sorted based on the number of occurrences). There are largely three categories that cause the failures: Infrastructure (IF), AI Engine (AE), and User (U). A failure reason may be observed in multiple categories.

the data collection period. The most critical one is *CPU out of memory*, where we see 185.7 as the `User` repetition factor. Interestingly, profiling shows that a certain engineer issued a number of training jobs, all of which suffer from the same out-of-memory issue, resulting in high concentration of failures. This motivates the need for runtime detection mechanisms that can correlate errors from the same user even though her jobs are independent from job management point of view.

User/programming errors lead to a lot of failures. Failures incurred by user errors, such as configuration/syntax/semantic errors in program and script, are dominant. These failures are very prevalent across our top-8 failure reasons. As explained, *CPU out of memory* is the most frequent with its failures significantly concentrated on a few users. Other frequent failures such as *incorrect inputs* and *semantic error* are more spread out across different users. From our profiling, the primary factor that causes those failures is a lot of independent components involved in a training job. For example, by definition, *incorrect inputs* happens when there is a failure in reading model or input data stored in external HDFS store. This is due to any error ranging from the data path is not correct, data format is inconsistent, data itself is corrupted on HDFS etc. Often, issues in data format affect multiple engineers in the same team (e.g., speech recognition team) as they often share the same training data or reference model.

4.2.3 Runtime to Failure

Column 4 of Table 7 presents runtime to failure (RTF) for each classified failure reason. To capture the summary of RTF distribution, in addition to the average, we also present the 50th-percentile (or median) and higher percentiles such as 90th-percentile and 95th-percentile.

The runtime to failure (RTF) exhibits high variability, with mainly short RTFs. Many failures of training jobs happen quickly, for example within 10 mins. This is mostly the case for failures driven by users in syntax, semantic, and configuration errors, which we can also infer from low 50P RTFs in the corresponding failure reasons. Note that most of those failures are deterministic and are caught when the runtime begins to execute the program. One of exceptions that is noteworthy is failure corresponding to inconsistent/corrupted input data. We can only detect this at the moment we actually read the erroneous data and attempt to parse it. This is the primary reason for having high 95P in *incorrect inputs*.

Infrastructure failures occur infrequently but have much longer runtime to failure (RTF). This analysis focuses on two failure reasons in infrastructure category: *model checkpoint error* and *MPI runtime failure*. They represent program-to-storage and program-to-program communication, which are both critical for reliable distributed deep learning training. In general, these errors are relatively infrequent compared to other common errors, constituting only 6.2% of the total `Trials`. However, our analysis shows that these errors tend to appear after a long execution duration, and thus dominate the time until failure detection. In particular, Table 7 shows that when the corresponding RTFs are summed up (i.e. `Total`), the two failure reasons, *model checkpoint error* and *MPI runtime error*, occupy as much as 21.73% and 14.63%, respectively.

4.2.4 Impact of GPUs Allocated

For jobs with the same RTF, the impact on the amount of utilized resources is proportional to the number of allocated GPUs. The larger the allocation, the bigger the impact.

Large jobs with programming semantic errors tend to fail a while after execution. Column 5 of Table 7 presents GPU demand across failure reasons. To simplify the analysis, we select four most-dominant failure reasons each contributing around 10% or more of failures. When we correlate RTF with GPU demand, among the four failure types, *semantic error* exhibits a markedly distinct trend, with jobs that have higher GPU demand having relatively large RTFs, as compared to jobs having lower GPU demand. This results in disproportional impact on the actual resources utilized by failed jobs. We show this in Column 6 of Table 7.

Column 6 presents actual GPU times for failures while multiplying RTF by GPU demand. As the results show, compared to the RTF only, the impact of *semantic error* increases up to 17.06% from 9.22% while the other three types of failure are either decreased or unchanged. This corresponds to the fact that semantic error is relatively frequent in larger-demand larger-RTF jobs. Looking deeper, we observe that training program instances sometimes send, receive, and access data in an inconsistent way during model parameters synchronization. As a consequence, *semantic error* ranks the second in terms of GPU resources used among failures in our workload.

5 Design Implications for Future Schedulers

Based on our experiences and data-driven analysis so far, in this section we discuss guidelines pertaining to the design of next-generation schedulers for DNN training workloads.

Prioritizing locality. One of the main results from our analysis of GPU scheduling was that lack of locality impacts both utilization and job running time. Our current scheduler adopts a classic approach where it waits for a limited time to see if locality can be achieved and if not the job is scheduled with the resources available at relaxed locality. The main reason for this approach is to keep queuing time low as longer wait times affect user experience.

However given that deep learning jobs run for many hours or even days, incurring a 10% or 20% drop in efficiency could extend the job running time by multiple hours. Thus in such scenarios, waiting for locality for a longer time could be more beneficial. However this requires inferring long-running jobs and appropriately setting user expectations. An alternate strategy could be to *migrate* a job to machines with better locality if resources become available during the execution.

Mitigating interference. Another critical guideline for schedulers would be to consider job placement policies to mitigate inter-job interference. Instead of packing *different* small jobs on a single server, one option would be place them on dedicated servers, reducing sharing and thus interference among such jobs. Such an option would increase fragmentation and will result in larger jobs having to wait for longer if we have to prioritize for intra-job locality. Support for job migration to *defragment* the cluster [52], especially applied to smaller jobs, will mitigate interference for small jobs, and will improve intra-job locality for large jobs.

Improving failure handling. A large number of job failures we see come from user errors in code or configuration. This is primarily because programming languages in use are typically not strongly typed. We have found that simple syntax checking could prevent many errors (e.g., missing quotes or parenthesis) and some of the more sophisticated runtime failures can be captured by running the first iteration of training. We plan to set up a pool of cheaper VMs to pre-run jobs. Even running multi-GPU jobs on a single GPU will catch such errors before they run on larger shared clusters and thus prevent *wasted* GPU cycles on them. Training failures also happen due to erroneous data format (e.g., inconsistent columns in samples). We plan to investigate having a well defined schema for datasets used in machine learning, and perform a schema check while accessing data to reduce such errors.

Another useful extension for multi-tenant GPU clusters would be to develop a system to predictively mitigate failures by proactively observing related failures. The main goal of such a system would be to (i) classify error messages in real time from logs that training jobs generate, and (ii) adapting scheduling parameters per job (e.g., number of retries) as well as across jobs (e.g., input data blacklisting) to reduce failure occurrences. For example, the scheduler could stop retrying for failure categories like incorrect data/model input and continue retrying for network timeouts.

6 Related Work

Failure analysis of data analytics jobs in shared clusters. Prior work has looked at designing large-scale data analytics platforms assuming that failures are common [10, 17, 50]. They focus on framework support for fault-tolerance and reliable job execution. In this paper, we focus instead on understanding job failures in deep learning specific platforms.

Kavulya *et al.* conducted a detailed characterization for job failures in a production MapReduce cluster [30]. Some of their findings include: (1) Many jobs fail within a few minutes while the worst-case job takes up to 4.3 days for its failure to be detected. These failures occur due to data copy errors and are similar to HDFS-related failures that we observe taking much longer to detect; (2) Many failures are related to either exceptions due to array indexing errors or IO exceptions. We again see some similarity to our work where coding errors lead to a number of failure cases.

Scheduler and runtime for efficient machine learning execution. SLAQ schedules concurrent machine learning

training jobs based on quality improvement for resource usage, allocating cluster resources for average quality improvement [56]. While this may improve the quality across jobs, each individual job may take longer time to finish. Optimus [38] leverages the convergence curve to predict job remaining time for dynamic resource scheduling and reduces average job completion time. It adopts an online fitting model to derive a proper number of servers and workers for MxNet [15] jobs in parameter server architecture. Tiresias [22] reduces job completion times when many training jobs undergo a *trial-and-error* exploration where job remaining time to complete training cannot be estimated from the convergence curve. In this work, we found that a large job experiences highly varying efficiency over placement spectrum (e.g., Table 5), and that future schedulers may need to consider the trade-off between reducing queueing time and reducing job running time more carefully over a wide range of locality choices.

We also note that an earlier technical report of our work [27] was used to motivate new scheduling primitives in recent work on scheduling like Gandiva [52]. More importantly, our paper presents a systematic study of a large-scale production cluster, covering the whole lifecycle of deep learning jobs including queuing, execution, and failure. We hope that our study of large clusters dedicated to deep learning workloads will continue to motivate novel research in deep learning platforms and schedulers for these workloads.

GPU resource management for machine learning. There are recent efforts on GPU sharing for simpler machine learning tasks. Baymax [14] explores GPU sharing as a way to mitigate both queuing delay and PCIe contention. Following that, Prophet [13] proposes an analytical model to predict performance of GPU workloads. Gandiva [52] proposes GPU time-sharing in shared GPU clusters through checkpointing at low GPU memory usage of training job. Future work includes integrating these prior work to improve cluster utilization and capacity to run more jobs.

Many training networks are memory bound, especially by capacity. Ryu *et al.* analyzed memory allocation for ImageNet [25], with recent VGG-16 model consuming up to 28 GB of memory [40]. Therefore, vDNN [40] proposes virtualized memory manager, and SuperNeurons [51] adopts fine-grained layer-wise memory control to schedule memory flexibly between CPU and GPU. Our work shares some similarity with prior findings (*i.e.*, some large networks do not fit in GPU memory) in real-world data.

Approximate data processing. Approximate data processing allows trading off accuracy for earlier completion times [6, 9, 16, 24, 28, 55]. In databases, online aggregation has been studied in the context of SQL queries [16, 24, 55]. More recently, approximation has been used in batch processing [6, 8, 49]. Machine learning training presents a fertile ground for exploring trading off accuracy for early completion. In this paper, for the training workloads run on our clusters, we quantify how trading off a very small amount of accuracy (0.1%) can result in significant savings in GPU execution time.

7 Conclusion

In this paper we analyzed a trace of deep learning jobs run on a large multi-tenant cluster of GPUs and studied various factors that affect cluster utilization. Our findings indicated the importance of locality for distributed training jobs and also how interference from other colocated jobs could lead to lower GPU utilization. We also performed a detailed analysis of various failure causes and showed how errors from various parts of the stack contribute to failures. Based on our data analysis and experiences running a large-scale operation, we also described guidelines that could help future research and development of machine learning schedulers.

Finally, we have publicly released the scheduler trace containing information about job arrivals, job size, placement and runtime to the community. As far as we know, this is the only trace that includes rich information about deep learning training jobs run in production. By making such a trace available, we hope to spur future research in this area.

Acknowledgments

We thank our shepherd, David Nellans, and the anonymous reviewers for their valuable comments and suggestions. We also thank Lidong Zhou, Chris Basoglu, Daniel Li, Ashish Raniwala, Swapnil Palod and the rest of the Microsoft Philly team for their unwavering help and support. This work was supported in part by NRF-2018R1C1B5086586.

References

[1] Deep Learning for Siri's Voice. https://machinelearning.apple.com/2017/08/06/siri-voices.html.

[2] Hadoop: Fair Scheduler. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[3] NVIDIA Management Library. https://developer.nvidia.com/nvidia-management-library-nvml.

[4] Using Deep Learning to Create Professional-Level Photographs. https://research.googleblog.com/2017/07/using-deep-learning-to-create.html.

[5] 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 *OSDI*, 2016.

[6] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[7] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys*, 2011.

[8] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *NSDI*, 2014.

[9] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic Sample Selection for Approximate Query Processing. In *SIGMOD*, 2003.

[10] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[11] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *OSDI*, 2014.

[12] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB*, 2008.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *ASPLOS*, 2017.

[14] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *ASPLOS*, 2016.

[15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[16] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online Aggregation and Continuous Query Support in MapReduce. In *SIGMOD*, 2010.

[17] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, February 2013.

[18] Dror G Feitelson. Packing schemes for gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer, 1996.

[19] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*, volume 1. MIT press Cambridge, 2016.

[20] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[21] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns. *CoRR*, abs/1707.03750, 2017.

[22] Juncheng Gu, Kang G. Chowdhury, Mosharaf abd Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[24] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *SIGMOD*, 1997.

[25] ImageNet, 2016. http://image-net.org.

[26] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Technical Report MSR-TR-2018-13, 2018.

[28] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Trans. Database Syst.*, 33(4):23:1–23:54, December 2008.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.

[30] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *CCGRID '10*, 2010.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[32] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[33] Matthew L Massie, Brent N Chun, and David E Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *Parallel Computing*, 30(7):817–840, 2004.

[34] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel Approximation Methods for Speech Recognition. *arXiv preprint arXiv:1701.03577*, 2017.

[35] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[36] Open Platform for AI, 2018. https://github.com/microsoft/pai.

[37] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[38] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *EuroSys*, 2018.

[39] PyTorch, 2018. https://pytorch.org/.

[40] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *MICRO*, 2016.

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *EuroSys*, 2013.

[42] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[44] Apache Hadoop 3.2.0 Submarine, 2019. https://hadoop.apache.org/docs/r3.2.0/hadoop-yarn/hadoop-yarn-applications/hadoop-yarn-submarine/.

[45] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM Neural Networks for Language Modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper With Convolutions. In *CVPR*, 2015.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NIPS*, 2017.

[48] 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 *SoCC*, 2013.

[49] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *OSDI*, 2014.

[50] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *EuroSys*, 2015.

[51] 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.

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.

[53] Hangchen Yu and Christopher J. Rossbach. Full Virtualization for GPUs Reconsidered. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2017.

[54] 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.

[55] Kai Zeng, Sameer Agarwal, and Ion Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *SIGMOD*, 2016.

[56] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: Quality-driven Scheduling for Distributed Machine Learning. In *SoCC*, 2017.

Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures

Erci Xu
Ohio State University

Mai Zheng
Iowa State University

Feng Qin
Ohio State University

Yikang Xu
Alibaba Group

Jiesheng Wu
Alibaba Group

Abstract

Modern datacenters increasingly use flash-based solid state drives (SSDs) for high performance and low energy cost. However, SSD introduces more complex failure modes compared to traditional hard disk. While great efforts have been made to understand the reliability of SSD itself, it remains unclear what types of system level failures are related to SSD, what are the root causes, and how the rest of the system interacts with SSD and contributes to failures. Answering these questions can help practitioners build and maintain highly reliable SSD-based storage systems.

In this paper, we study the reliability of SSD-based storage systems deployed in Alibaba Cloud, which cover near half a million SSDs and span over three years of usage under representative cloud services. We take a holistic view to analyze both device errors and system failures to better understand the potential casual relations. Particularly, we focus on failures that are **R**eported **A**s "**S**SD-**R**elated" (**RASR**) by system status monitoring daemons. Through log analysis, field studies, and validation experiments, we identify the characteristics of RASR failures in terms of their distribution, symptoms, and correlations. Moreover, we derive a number of major lessons and a set of effective methods to address the issues observed. We believe that our study and experience would be beneficial to the community and could facilitate building highly-reliable SSD-based storage systems.

1 Introduction

Flash-based solid state drives (SSDs) have become an indispensable component of modern datacenters due to its superior performance and low power draw [6]. Various applications, including databases [14], social network [15], and online shopping [41], have been supported by large-scale SSD-based storage systems. Therefore, the reliability of such systems is of critical importance.

However, it is challenging to maintain the high reliability of SSD-based storage systems. First, unlike hard disk drives (HDDs), SSDs may experience unique flash errors (e.g. program errors [20, 38]) which are sensitive to the environment (e.g., temperature [30]). Therefore, our decades of collective wisdom on HDDs is not fully applicable. Second, issues in the traditional HDD-based storage stack (e.g., faulty interconnection and human mistakes [28]) may continue to haunt SSD-based storage systems. In addition, due to the complexity of storage systems, the potential correlations among various events across different levels/components are not well-understood, rendering extreme difficulty in pinpointing the root causes of system failures or comping up with effective fixes.

To address the challenges, substantial efforts have been made to understand the reliability of SSD itself [24, 33, 38, 44]. For example, Schroeder et al. [38] study flash errors and discover correlations between flash errors and other device attributes (e.g., age, wear, lithography). Zheng et al. [44] analyze the behavior of SSDs under power faults. Narayanan et al. [33] analyze a diverse set of device factors (e.g., design and provisioning) and their correlations with failed SSDs. Hao et al. [24] study the performance instability involving millions of drive hours, especially the device latency in RAID groups. While these studies provide valuable insights on the characteristics of SSDs, it remains unclear how SSDs interact with the rest of the system and contribute to system failures.

Besides the work on SSDs, studies on HDD-based storage systems are also abundant [7, 8, 28, 35, 37]. Apart from understanding HDD errors in the field [7, 35, 37], researchers analyze the correlations between HDD errors and system failures [8, 28]. However, since SSD-based systems are significantly different from HDD-based systems (e.g., the TRIM command support throughout the OS kernel [2]), it is unlikely that these studies and findings are directly applicable to SSD-based storage systems.

In this paper, we look into the storage systems deployed in 7 datacenters of Alibaba Cloud, which includes around 450,000 SSDs over 3 years' deployment. Similar to other large-scale deployed systems [16, 17, 29, 42], our target systems are equipped with system monitoring daemons de-

ployed on each node of the clusters. The daemon monitors abnormal behaviors by constantly checking the BIOS messages at boot time, the kernel syslog at runtime, and the functionality and availability of the cloud services. Upon an abnormal event, the daemon will report a failure ticket with the timestamp, the component involved, and a snippet of corresponding logs.

Among all failure tickets, we focus on failures that are **R**eported **A**s "**S**SD-**R**elated" (**RASR**) in this paper. Also, we collect the corresponding repair logs of the failures as well as the SMART [4] logs of the SSDs involved. By holistically analyzing the three datasets (i.e., failure tickets, repair logs, and SMART logs) in the context of the storage systems design and deployment, we identify a number of interesting characteristics of RASR failures in terms of distributions, symptoms, and correlations. Moreover, we perform field studies and validation experiments to understand in depth the factors affecting RASR failures, and to derive a number of major lessons as well as realistic remedies for hardware architects, software engineers, and system administrators. More specifically, our contributions include the following:

(1) Characteristics of RASR Failures. We collect about over 150K failure tickets in total from the target systems. Among these failure tickets, we find that 5.6% are RASR failures (i.e., about 10K instances), which manifested in five symptoms: *Node Unbootable*, *File System Unmountable*, *Drive Unfound*, *Buffer IO Error*, and *Media Error*. By correlating the RASR failures with the repair logs, we find that a significant number (34.4%) of RASR failures are *not* caused by the SSD device. For example, plugging SSDs into wrong drive slots, a typical *human mistake*, accounts for 20.1% of RASR failures. Moreover, for RASR failures caused by SSDs, we find that both the location of devices (i.e., in different datacenters) and the type of cloud services may affect SSD failure rates.

(2) Lessons and Actions for Hardware Architects. We find that the suboptimal intra-node SSD stacking and intra-rack node placement can lead to *passive heating* (i.e., heating on *idle* SSDs by neighboring active SSDs), which may in turn cause a large number of device errors and high failure rates. Moreover, by experimenting on a dedicated cluster with continuous temperature monitoring, we are able to verify that the poor rack architecture can increase the temperature of *idle* SSDs by up to 28 C°, resulting in 57% more device errors after 128 hours of passive heating.

To reduce the impact of passive heating, we formulate a new strategy for intra-rack node placement. Furthermore, we propose a proactive approach to alleviate the passive heating by routinely scanning the entire device to trigger the FTL internal read refresh [11]. Different from the traditional data scrubbing [5, 31], the scanning is lightweight enough to be scheduled more frequently to reduce the effect of passive heating. Our results show that performing a scanning every 4 hours can offset most negative impact of passive heating. Although not observed in our experiments, the scanning may potentially lead to more read disturbs [10], affecting the device negatively. Therefore, we believe it would be ideal for the vendors to implement the proactive scanning at the FTL.

(3) Lessons and Actions for Software Engineers. We find that both the data allocation scheme in the service software stack and the I/O pattern of cloud services play important roles in affecting SSD reliability and leading to RASR failures. For example, the Block service, empowered by a direct mapping based data allocation scheme, can cause severe imbalance of SSD usage when running on top of an HDFS-like distributed file system (DFS): 15-20% SSDs are overly used, which causes up to 77.3% more device errors and up to 18.7% higher device failure rate. Inspired by the log-structured file system [36], we optimize the data allocation scheme on the target systems by adding a shared appending log, and thus mitigate the imbalance issue.

(4) Lessons and Actions for System Administrators. By co-analyzing device-level and system-level logs, we discover a strong correlation between one type of RASR failures (i.e., those caused by faulty interconnection) and one type of device errors (i.e., Ultra-DMA CRC or UCRC). Based on this observation, we design an indicator for the faulty interconnection issue based on the accumulation of UCRC errors, which significantly improves the repair procedure of relevant failures. In addition, we find that SSDs on the target systems serve three different purposes (i.e., system drives, storage, and buffering), but they all use the same SATA interface. This causes much confusion for system administrators who need to replace drives. To reduce the chance of plugging SSDs into wrong drive slots (cause of 20.1% RASR failures), we adapt the systems to use different SSD interfaces for different purposes (e.g., U.2/M.2 for system drives and SATA for storage). This optimization effectively eliminates the confusion and reduce the corresponding failures.

To the best of our knowledge, our work is the first effort on understanding the characteristics of RASR failures as well as the causal relation between SSD errors and the system design and usage, in large-scale production systems. Based on this study, we have significantly improved the reliability of practical systems through a number of simple yet effective mechanisms (e.g., proactive data scanning, UCRC-based indicator and specializing interfaces). We believe that our study and lessons would be beneficial to the community, and could facilitate building highly-reliable SSD-based storage systems.

The rest of the paper is organized as follows: §2 introduces our methodology; §3 analyzes the characteristics of RASR failures; §4 - §6 discusses our lessons and actions for hardware architects, software engineers, and system administrators, respectively; §7 discusses related work, and §8 concludes the paper.

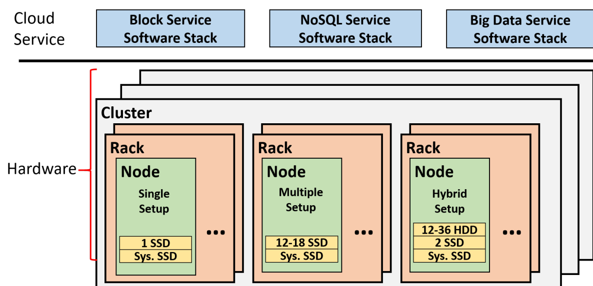


Figure 1: Architecture of target systems.

Model	Capacity	Lithography	Age	Vendor
M1	480 GB	20 nm	2-3 yrs	A
M2	800 GB	20 nm	2-3 yrs	A
M3	480 GB	16 nm	1-2 yrs	A
M4	480 GB	20 nm	2-3 yrs	B
M5	480 GB	20 nm	1-2 yrs	C

Table 1: Characteristics of SSDs in the target systems.

2 Methodology

2.1 System Architecture

We study SSD-based large-scale storage systems deployed in 7 datacenters. The architecture of the target systems is shown in Figure 1.

At the device level, the systems include around 450,000 SSDs spanning three years of deployment. As shown in Table 1, these SSDs cover a spectrum of variability in terms of capacity, lithography, age, and vendors. Note that all five models in our dataset are using SATA interfaces and based on MLC NAND cells.

Each node in the systems employs one of three different setups of SSDs: (1) *Single*: a node contains one SSD for storing temporary data; (2) *Multiple*: a node contains 12 to 18 SSDs for persistent storage; (3) *Hybrid*: a node contains 2 SSDs and 12 to 36 HDDs where the SSDs are used for buffering incoming writes. In addition, each node has one SSD serving as the system drive.

A rack consists of 16 to 48 nodes, and a DFS cluster spans 12 to 18 racks. On top of the DFS, the system supports three types of cloud services, including Block service, NoSQL service, and Big Data service. As shown in Table 2, the cloud services may run on different setups where the SSDs are used for different purposes.

2.2 Raw Datasets

The target systems include sophisticated monitoring mechanisms, similar to other large-scale systems [16, 17, 29, 42]. The monitoring daemons are deployed on each node of the clusters, and they log various events either periodically or upon the occurrence of an event.

At the system level, the daemons monitor BIOS messages,

Service	SSD Model	Setup	Usage
Block	all models	Hy/Mul	Pers/Buf
NoSQL	M1, M3, M4, M5	Hy/Mul	Pers/Buf
Big Data	M1, M2, M4	Single	Temp

Table 2: Cloud services and SSD usages. ***Hy****: Hybrid;* ***Mul****: Multiple;* ***Pers****: Persistent storage;* ***Buf****: Buffering writes;* ***Temp****: Temporarily storing intermediate data.*

kernel syslogs, and the service-level verification of data integrity. Upon an abnormal event, the daemon reports a failure ticket with the timestamp, the related hardware component, and a log snippet describing the failure. Each failure ticket is tagged based on the component involved. For example, if an SSD appears to be missing from the system, the failure ticket is tagged as ”SSD-related”. If there is no clear hardware component recorded in the logs, the ticket would be tagged as Unknown.

At the device level, the daemons record SMART attributes [4] on a daily basis, which cover a wide set of device behaviors (e.g., total LBA written, uncorrectable errors).

Besides the failure tickets and SMART logs, we collect the repair logs of all RASR failures, which are generated by onsite engineers after fixing the failures. For each failure event, the corresponding repair log records the failure symptom, the diagnosis procedure, and the successful fix.

2.3 Study Approaches

After collecting the failure tickets, the SMART logs, as well as the repair logs, we apply the following approaches to derive insights:

- **Log analysis:** We calculate the distributions of failure events along multiple dimensions (e.g., hardware types, manifestation symptoms). Moreover, since the number and the variety of events in the logs is large, we leverage classic statistical algorithms (e.g., Spearman Rank Correlation Coefficient [12]) to analyze the characteristics of individual events as well as the potential correlations among different events.
- **Field studies:** Besides the log analysis, we visit the datacenters in person to investigate the potential variance of target systems in terms of cluster architectures, which turns out to be critical for discovering the passive heating phenomenon (§4). Also, we discuss with on-site engineers to empirically verify our hypothesis on RASR failures.
- **Validation.** We build a dedicated cluster to validate our hypothesis. Moreover, to address the issues exposed in our study, we design a set of remedy methods, and validate the effectiveness and practicability on production systems.

2.4 Limitations

Failure Reporting. Our study relies on the failure tickets reported by distributed daemons that automatically monitor the health condition of system components from hardware to software. The daemons may fail to record (e.g. network

RASR Failure Symptom	Meaning	Distribution
Node Unbootable	Unable to boot the OS on a node	2.6%
File System Unmoutable	Unable to mount a local file system	7.4%
Drive Unfound	A device cannot be found by the system software	53.7%
Buffer IO Error	Unable to write data from memory buffer to the device	17.3%
Media Error	Unable to read correct data from the device	19.0%

Table 3: Distribution of RASR failures based on manifestation symptoms. *This table shows five different symptoms of RASR failures and the corresponding percentage.*

Hardware Type		Distribution
CPU		0.7%
Memory		8.5%
Network		34.0%
Motherboard		5.4%
Storage	HDD	22.1%
	SSD	5.6%
Unknown		23.7%

Table 4: Distribution of failure tickets based on hardware types. *This table shows the distribution of failure tickets that are tagged as related to major hardware components.*

RASR Failure Symptom	Affected Rate (‰)				
	M1	M2	M3	M4	M5
Node Unbootable	0.24	0.42	0.15	0.13	0.07
FS Unmountable	1.28	1.05	0.42	2.90	2.04
Drive Unfound	11.19	8.58	5.31	11.51	4.38
Buffer IO Error	3.73	1.34	1.36	4.06	1.21
Media Error	3.42	5.24	2.81	5.73	1.33

Table 5: Distribution of RASR failures among five SSD models. *This table shows the affected rate of each SSD model (M1-M5), which is the number of SSDs involved in one type of RASR failures divided by the total number of SSDs with the same model.*

failure during log collection) or inaccurately tag the events (e.g. a node crash tagged as "Unknown" due to insufficient logs). However, to the best of our knowledge, the way the tickets are reported is the common practice widely used in major large-scale production systems and previous studies on large-scale deployed systems also rely on similar mechanisms for collecting datasets [16, 17, 29, 42].

Software Stack Design. Our software stack includes OS, DFS and service components. Apart from using a major distribution of Linux, our DFS and service software are not open-source. Nonetheless, they share generic similarities with popular large-scale storage systems such as HDFS [39] and Google File System [19], and similar high-level services are provided by other companies such as EBS [1] and DataStore [3].

Hardware Products. Like previous works [32, 33], the target systems use off-the-shelf hardware products such as SSDs and interconnects. Many products are also widely deployed in the datacenters of other organizations. Therefore, users from other organizations may encounter the same or similar hardware-related issues, and we hope they can benefit from our experiences.

3 Characteristics of RASR Failures

3.1 Overview of Failure Tickets

We collect all failure tickets reported as related to hardware components, over 150K tickets in total. Table 4 shows the distribution of the failure events based on the types of hardware components involved, including CPU, Memory, Network, Motherboard, HDD/SSD, and Unknown. The Unknown type refers to the failures where a relevant component is not specified in the daemon-reported ticket. The second column shows the percentage of failure events for each type of hardware. According to our daemon setup, no failure event is tagged with more than one type.

As shown in Table 4, storage components (i.e., HDD and SSD combined) contribute to 27.7% (i.e., 22.1% + 5.6%) of all hardware-related failure events. RASR failures alone account for 5.6%. Compared with other hardware components (e.g., Network which accounts for 34.0%), RASR failures are much fewer in our dataset. This is consistent with the findings from previous studies that SSD is a relatively reliable component among all hardware components deployed in datacenters [6, 33].

Nonetheless, since the total number of failure events is large (i.e., over 150K), even a relatively small percentage (i.e. 5.6%) of failures cannot be ignored. Therefore, we perform an in-depth analysis on RASR failures in this study and present detailed results in the following sections.

3.2 Symptoms of RASR Failures

After analyzing all RASR failure logs, we find that RASR failures can manifest in multiple ways. As shown in Table 3, there are five different types of manifestation symptoms, including *Node Unbootable, File System Unmountable, Drive Unfound, Buffer IO Error,* and *Media Error*. The meaning of each symptom is described in the second column of the table. Also, the distribution of each type of symptoms is listed in the last column of Table 3.

Among the five symptoms, the Drive Unfound type, which means the device cannot be found by the system software, is the dominant one (i.e., accounts for 53.7%). Based on

Node Unbootable	File System Unmountable	Drive Unfound	Buffer IO Error	Media Error
1.Slot Check(53.8%) 2.Repl. SSD(46.2%)	1.Mnt. Opt. Check(5.4%) 2.FSCK(40.5%) 3.Repl. SSD(54.1%)	1.Rebooting(22.2%) 2.Slot Check(34.8%) 3.Repl. Cable(25.9%) 4.Repl. SSD(16.1%)	1.FSCK(79.8%) 2.Repl.SSD(20.2%)	1.Data Check(30.2%) 2.Repl. SSD(69.8%)

Table 6: Repairing procedures of RASR Failures and their successful rates grouped by symptom. *The first row shows five manifestation symptoms of RASR failures. The 2nd row lists repairing procedures for each symptom. The repairing follows an order as indicated by the number before each fix approach. The rate in the parentheses after each fix indicates within that symptom group the percentage of failures fixed by that approach. Repl.: replacing; Mnt. Opt.: Mount Options.*

RASR Failure Symptom	Affected Rate (‰)		
	Block	NoSQL	BigData
Node Unbootable	0.27	0.12	0.35
FS Unmountable	1.43	1.05	1.42
Drive Unfound	13.25	10.58	9.31
Buffer IO Error	5.73	2.34	5.36
Media Error	8.42	3.24	3.77

Table 7: Distribution of RASR failures among cloud services. *This table shows the affected rate of each cloud service (i.e. Block service, NoSQL service and Big Data service), which is the number of M1 SSDs involved in one type of RASR failures divided by the total number of M1 SSDs within the same cloud service.*

RASR Failure Symptom	Affected Rate (‰)				
	DC1	DC2	DC3	DC4	DC5
Node Unbootable	0.35	0.31	0.21	0.27	0.23
FS Unmountable	1.08	1.25	1.42	1.90	1.04
Drive Unfound	10.33	12.72	13.31	13.96	14.10
Buffer IO Error	2.95	2.14	1.98	1.86	2.12
Media Error	2.06	3.04	2.85	**7.73**	3.75

Table 8: Distribution of RASR failures among datacenters. *This table shows the affected rate of each M1 SSD under the Block service from 5 datacenters (DC1-DC5), which is the number of M1 SSDs involved in one type of RASR failures divided by the total number of M1 SSDs under the Block service within the same datacenter.*

our discussion with on-site engineers, a Drive Unfound event may be masked by the system software (e.g., automatic redirection of I/O requests and re-replication of data), and may not necessarily lead to data loss. However, the event can still cause additional latency on the I/O requests involved, and usually requires engineers to diagnose the issue on site. Similarly, other types of RASR failures may also affect system performance and consume manual efforts. Therefore, it is important to understand the root causes of RASR failures and improve the failure handling. We discuss the analysis on fix procedures in §3.3.

After observing the distribution of RASR failure symptoms, we further study the correlation between RASR failure symptoms and other important factors, including SSD models, service workloads, and datacenter locations.

As mentioned in Table 1, there are five different SSD models in our target systems. To further understand the potential impact of SSD models on RASR failures, we calculate the failure affected rate for each model, which is the number of SSDs involved in one type of RASR failures divided by the total number of SSDs with the same model. As summarized in Table 5, the five RASR failure symptoms have been observed on all five SSD models (M1-M5). The affected rate ranges from 0.07‰ (i.e., M5 SSDs with the Node Unbootable symptom) to 11.51‰ (i.e., M4 SSDs with the Drive Unfound symptom). We do not observe statistically significant difference among SSD models in terms of the affected rate of RASR failures, which suggests that RASR failures may not be directly related to the characteristics of SSD models.

To study the correlation between RASR failures and service workloads running on the target systems, we use M1 SSDs, a popular model accounting for 35% of the drive population. Table 7 shows the affected rates of M1 SSDs under three cloud services. We observe that the Block service (2nd column) has the highest affected rates in four out of five types of RASR failures (except Node Unbootable). This finding motivates us to further investigate the cloud services with their designs, drive usage and device level errors in §5.

In addition, we study whether the location (i.e. datacenters) plays a role in RASR failures. We evaluate the affected rates of M1 SSDs under the Block service (i.e. the main service accounting for for 57% of SSD deployment) in different datacenters (DCs). Table 8 summarizes the results. Note that M1 SSDs of the Block service are only used in five datacenters, i.e., from DC1 to DC5. From the table, we observe that while no DC dominates all failure types, DC4 has substantially more Media Errors (i.e. last row), indicating more data corruptions. To better understand the potential root causes, we study the uniqueness of DC4 in terms of hardware architectures, especially the SSD placement in §4.

3.3 Fixes of RASR Failures

To understand the potential root causes of RASR failures, we further analyze the corresponding repair logs. For each failure, administrators apply a symptom-based repairing procedure, i.e., trying a pre-defined sequence of fix candidates one by one based on the failure symptom until the failure dis-

appears. Each repair log records the repairing process and the successful fix of a failure event. Table 6 summarizes the pre-defined sequence of fix candidates for each RASR failure symptom. Also, for each fix candidate, we calculate its successful rate in the group of failures with the same symptom (shown in the parentheses).

For instance, after observing a Drive Unfound failure (3rd column of Table 6), administrators will first attempt to remotely reboot the node to check whether the failure is transient ("Rebooting"). If not, administrators will manually check whether the device is plugged into the correct slot ("Slot Check"). If the slot is correct, administrators will then try replacing the cable ("Repl. Cable"), followed by replacing SSD ("Repl. SSD") as a final resort until the failure is resolved. Note that all RASR failures are eventually fixed by replacing SSDs if previous attempts do not work.

One observation on Table 6 initially puzzling us is that the first fix attempt is not always the most effective one within each group of failures. For example, "Mnt. Opt. Check" (Mount Options Check) works only for around 5% of File System Unmountable failures (2nd column). Similarly, "Data Check" cures 30.2% of Media Error events (last column). After discussing with administrators, we realize that the symptom-based repairing procedure overall is simple yet effective. Specifically, the order of the fix candidates for each failure symptom is first based on their costs, followed by their effectiveness. As a result, the set of software-based fixes (i.e., checking mount options, rebooting, FSCK, and data check) are always preferred over the set of manual or hardware-based ones (i.e., slot check, replacing cable, and replacing SSD). The order within either set of fix candidates is based on their effectiveness to solve the failure symptoms in administrators' past experiences. The sequence of fix candidates for repairing Drive Unfound (3rd column) clearly demonstrates the ordering consideration.

Although existing fix procedure is effective to certain degree, it is a black-box approach (trail-and-error) since the administrators do not know the root causes before applying the fix candidates. This motivates us to conduct in-depth study on the potential root causes of RASR failures for helping system administrators with better fix strategy. As will be discussed in §6, we identify an accurate indicator for one type of failures (§6.1) and propose a method for avoiding another type of failures (§6.2).

3.4 SMART Logs under RASR Failures

The device level SMART [4] log is an important dataset for analyzing SSD behaviors and failures in the field [32, 33, 38]. Similar to previous studies [32, 33, 38], we analyze a subset of SMART attributes (as shown in Table 9) on our target systems in depth and observe a number of characteristics which are consistent with the prior work (e.g., the prevalence of uncorrectable errors and the high raw bit error rate on failed drives). Due to space limit, we do not discuss the observations or the distribution of SMART attributes that are similar to prior work. Instead, we correlate the SMART logs with RASR failures in later sections and analyze the impact of different factors (e.g., hardware architecture and software design) on drive behaviors.

Device Level Event	Definition
Host Read	Total amount of host LBA read from SSD
Host Write	Total amount of host LBA write to SSD
Program Error	Total # of errors in NAND programming operation
Raw Bit Error Rate (RBER)	Total bit corrupted divided by total bits accessed
End-to-End Error (E2E)	Total # parity check failures between drive and host
Uncorrectable Error	Total # of data corruption beyond ECC's ability
UDMA CRC Error (UCRC)	Total # of CRC check failures during Ultra-DMA

Table 9: Device level events collected in our study.. *Device level events are collected via SMART [4]. All events are recorded in a cumulative manner.*

4 Lessons & Actions for Hardware Architects

During our characteristics study of RASR failures (§3), we observe that SSDs deployed in one particular datacenter (DC4) experience much more Media Error under the Block Storage service (Table 8). Moreover, these SSDs have higher Raw Bit Error Rate (RBER) and Uncorrectable Bit Error Rate (UBER) based on the SMART logs.

To understand why the Block service in DC4 is so unique, we perform field studies at DC4 and other datacenters. We find that there are two potential factors. First, in DC4, about 27.1% Block service nodes are equipped with 18 SSDs, while in other datacenters less than 5.3% Block service nodes have 18 SSDs (most nodes have 12 SSDs). Second, in DC4, nodes for different services are often co-located in the same rack, while in other datacenters a rack is exclusively used for a single service. In this paper, we refer to the two factors as *intra-node SSD stacking* and *intra-rack node placement*, respectively, both of which affect the SSD placement in the systems. Since NAND flash memory is known to be less reliable under higher temperature [9] due to the Arrhenius Law[34], we suspect that the SSD placement may affect the airflow in nodes and racks, which may in turn affect the operating temperature of neighboring SSDs, and then lead to abnormal behaviors. We refer to this hypothesis as *passive heating*.

Note that passive heating is different from heating mechanisms used in prior work for analyzing NAND flash or

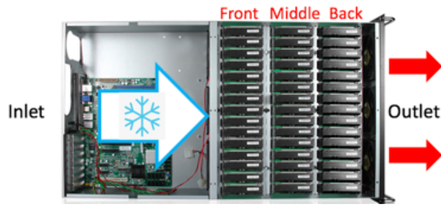


Figure 2: Intra-node SSD stacking and the airflow. *This figure shows the stacking of SSDs within a node, which include three groups:front, middle and back. The arrows indicate the direction of the airflow for cooling. Note that due to confidentiality, we cannot show the real photo of the node deployed in our target systems; however, the actual node is very similar to this example.*

SSD under high temperatures. Specifically, in the studies on NAND retention errors, NAND chips are heated to high temperature without power supply (e.g., heating in the oven) [9]. Differently, SSDs in our study are always powered on, where the FTL may proactively reduce NAND errors. In previous studies on the impact of SSD temperatures, they mostly focus on active heating, i.e., heating the SSDs by *heavily accessing* them. In such case, high temperature may trigger the throttling mechanism in FTL to reduce errors [32]. On the contrary, the passing heating we observe may affect *idle* SSDs, which cannot be remedied by throttling (because there is no heavy on-the-fly flash operations to throttle). Therefore, we believe it is necessary to investigate the passive heating further.

4.1 Identify and Verify Passive Heating

With the help of on-site engineers, we identify three potential scenarios where SSDs may suffer from excessive passive heating:

- **Hot Airflow.** Figure 2 shows an example of stacking of multiple SSDs within a node and the airflow for cooling. In this design, idle SSDs at the outlet of the airflow may be heated up when the front SSDs are being accessed heavily.
- **Hot Neighbors.** If an idle node is close to another node running intensive workloads, SSDs in the idle node may be heated up by the hot neighboring node.
- **Hot Air Recirculation.** When a node is removed out of the rack, the empty node slot may serve as a channel for tunneling hot airflow and passing heat to nearby nodes (one empty node slot away).

To verify and measure the passive heating, we build an experimental cluster with continuous monitoring of SSD temperatures and controlled workloads. The cluster includes 8 nodes in a dedicated rack, and each node has 18 SSDs. We perform the following experiments to analyze the passive heating in each of the aforementioned scenarios.

For Hot Airflow, we first record the initial temperature of the SSDs near the outlet of the airflow when a node is just powered on. Then, we run intensive workloads to access the 6 front SSDs (i.e. SSDs close to the inlet of the airflow), but leave the remaining 12 SSDs idle. We compare the temperatures of the idle drives before and after running the workloads.

For Hot Neighbors, we run intensive workloads on some nodes, and keep monitoring the temperatures of the SSDs on the neighboring idle nodes. We try three configurations where the hot neighbor(s) is atop, below, or are at both sides of the idle node.

For Hot Air Recirculation, we remove a node from the rack and examine whether the temperature of the SSDs of an idle node can be affected by a hot neighbor that is one node slot away.

Our experiments show that for an idle SSD initially at 25 C°, it can be heated up by 23 C°, 9 C°, and 17 C° (i.e., reaching 48 C°, 34 C°, and 42 C°) via Hot Airflow, Hot Neighbors, and Hot Air Recirculation, respectively. Moreover, when combining the three effects, an idle SSD can be heated up by 28 C° (i.e., reaching 53 C°) on our cluster.

4.2 Effects of Passive Heating on SSDs

After verifying that the suboptimal SSD placement may generate undesirable passive heating on SSDs, we look into the impact of passive heating on SSDs' behavior. In this set of experiments, we compare the raw bit errors of SSDs at three levels of temperatures under passive heating: 35C°, 45C°, and 55C°. Note that we use a fixed temperature interval (i.e., 10C°) to make the correlation between errors and passive heating more clear.

Specifically, in each experiment, we heat up idle SSDs (initially 25C°) through passive heating until they reach one of the three levels of higher temperatures, i.e., 35C°, 45C°, or 55C°. At each level, we maintain the same temperature for a range of time durations, i.e., from 1 to 128 hours, by carefully adjusting the workloads on neighboring nodes based on the feedback of the measured SSD temperature. After the stable passive heating period finishes, we scan the whole device and measure the Raw Bit Errors[1] newly generated during the heating period.

Figure 3 summarizes the results. We find that all three levels of passive heating (i.e., 35C°, 45C°, and 55C°) may lead to more Raw Bit Errors compared with normal case (i.e., 25C°). Additionally, a higher level of passive heating (e.g., 55C°) for a longer period of time (e.g., 64 hours) can generate more Raw Bit Errors, and the increasing trend is non-linear. Moreover, after 128 hours of heating, we observe that idle SSDs suffer from 57% more Raw Bit Errors.

Note that our observation in this set of experiments (i.e., higher temperature leads to more retention errors) aligns well with previous studies and industry standards[27, 32]. However, it contradicts to a recent study on 3D NAND flash chips [30]. This is likely because the structure and characteristics of 3D NAND are different from those used in our systems

[1] We do not use the Raw Bit Errors Rate (RBER) attribute directly because it is a cumulative value over the entire lifespan of a device.

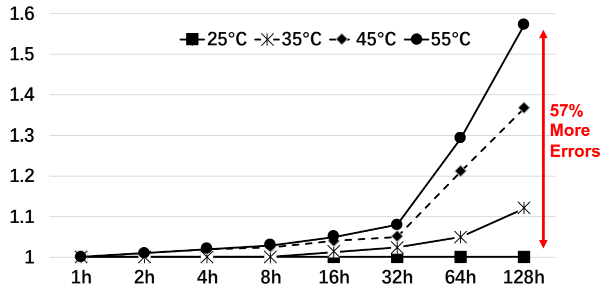


Figure 3: Raw Bit Errors generation under passive heating through time

(e.g., charging trap versus floating gate).

Although in our small scale experiments we do not observe uncorrectable errors or RASR failures, we believe that the results (e.g., increasing Raw Bit Errors) indicate that SSDs may become less reliable due to passive heating, and the phenomenon deserves more attention.

4.3 Offset the Impact of Passive Heating

Since the idle SSDs that are suffered from passive heating do not serve any I/O (before measuring the Raw Bit Errors), the increased Raw Bit Errors are most likely due to a retention issue. Classic techniques like data scrubbing can effectively mitigate retention issue by scanning and checking data integrity. However, it is unrealistic to apply such techniques frequently due to the prohibitive performance overhead.

On the other hand, we realize that the FTL in SSDs usually has a mechanism called Read Refresh [11] to correct bit errors and reallocate data during reading. So we propose to apply a lightweight regular software-based scanning to trigger read refresh (without computing checksums) to offset the negative impact of passing heating on idle SSDs. To verify our proposed method, we experiment on different intervals of scanning (e.g., 1 to 128 hours) and measure the reduction of Raw Bit Errors. Our experimental results are very promising: a routine scanning of every 4 hours can effectively control Raw Bit Errors without incurring too much overhead in our target systems. For example, after we perform a 4-hour routine scanning on the idle SSD during its 128 hours of passive heating under 55 C°, we only observe 1% more Raw Bit Errors, which is in stark contrast to 57% more Raw Bit Errors without scanning. Further increasing the frequency of scanning do not reduce the errors much. Therefore, the 4-hour-scanning routine achieves a good balance between the effectiveness and overhead in our systems.

While triggering read refresh by routine scanning is helpful for offsetting the impacts from passive heating, there are other potential issues with its direct deployment on production systems. First, the routine scanning requires fine-grained temperature monitoring to detect passive heating. Currently, the SSD temperature on our target systems is obtained by querying the SMART logs. Similar to other cloud companies [32, 33], the SMART logs are pulled on a daily basis in our production systems, which is insufficient for monitoring passive heating. Increasing the query rate requires changes to the distributed monitoring daemons and may affect the quality of service. While some hardware-based temperature querying methods (e.g., IoT sensors [18]) are relatively lightweight, integrating them into production systems may require significant efforts.

Second, the scanning might introduce more read disturb errors [10]. Although the scanning does not necessarily read the entire disk (i.e. only the stored data) or blindly get executed every 4 hours (i.e. only when SSD is in passive heating for more than four hours), the SSD may still suffer from increasing device errors due to read disturbance. This may further deteriorate as the lithography becomes smaller. Therefore, while effective, it is difficult to directly apply the routine scanning used in our experiments to production systems.

Alternatively, it is possible to implement our proposed technique of detecting and remedying passive heating in FTL with vendors' support. First, many SSDs today support heat throttling in the FTL, which implies that the temperature is already closely monitored by the device. Second, the FTL has the best knowledge of which parts of the data have higher error rates, and thus can react accordingly by proactively read refreshing the corresponding data. Therefore, the FTL-based solution may be more effective. We hope our study can raise the awareness of passive heating and facilitate addressing the issue.

5 Lessons & Actions for Software Engineers

As shown in Table 7, the SSDs under the Block service suffer more RASR failures than the devices under the other two services. This finding motivates us to further investigate the behavior differences of the SSDs among the three services, as well as the potential causes and fixes.

5.1 Usage Imbalance in Block Service

We start with the SSD usage, the most fundamental statistics of device behaviors. The three cloud services (i.e. Block, NoSQL, and Big Data Analytics) supported by our target systems are intrinsically different in terms of data placement policies and I/O patterns, which may lead to different usage patterns of SSDs. To understand the basic usage, we compare two device-level events: *Host Read* and *Host Write*, which measure the amount of data read from or written to the device by the host.

More specifically, we measure the hourly average value of host read/write (i.e., total sizes of host read/write divided by total power-on hours) on all SSDs under each cloud service. Moreover, we calculate the variability of the two metrics among SSDs under the same service using the coefficient of variation (CV), which is the ratio of standard deviation to

		Host Read	Host Write
Avg. Value /Hour	Block	7.69 GB	6.56 GB
	NoSQL	6.10 GB	5.28 GB
	BigData	1.57 GB	1.22 GB
CV	Block	35.5%	24.9%
	NoSQL	3.2%	6.2%
	BigData	1.8%	3.7%

Table 10: Comparison of SSD usages under three services in terms of host read and host write. *CV: Coefficient of Variance, the ratio of standard deviation to mean.*

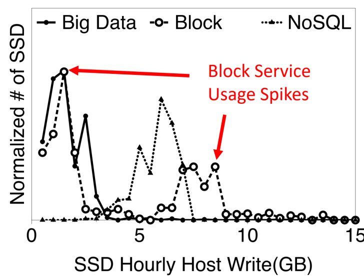


Figure 4: Distribution of SSDs under three services. *This figure shows the distribution of SSDs in terms of hourly host write under three services. The arrows mark the bimodal usage under the Block service.*

mean. Intuitively, a higher CV indicates that the hourly host read/write varies more across SSDs.

Table 10 summarizes the results. We can see that the hourly average value of host read and host write of the Block service are 7.68 GB and 6.56 GB, respectively, which are similar to those of the NoSQL service. However, the Block service has much higher variances for the two metrics (i.e., 35.5% and 24.9%), which implies that the usage of SSDs under this service is much more unbalanced.

Figure 4 further illustrates the distribution of SSDs in terms of hourly host write under the three services by using a histogram with 0.5 GB buckets along the x-axis. Each dot on the line (e.g., solid line for Big Data) represents the cumulative count of SSDs in the corresponding usage bucket. We can see from the figure that the majority of SSDs under NoSQL and Big Data Analytics services have similar usages (i.e., one major spike on the corresponding curve). In contrast, the SSDs under Block Storage service shows bimodal usages (i.e., two spikes far apart) as marked in the figure. Further analysis shows that the overly used drives (i.e. the right spike) account for around 17% of all SSDs in the Block Storage service and have 227.1% more write usage. The distribution of SSDs in terms of hourly host read exhibits similar pattern.

With such an unbalanced usage pattern, the overly-used set of SSDs may be worn out quickly. As a result, compared with averagely-used SSDs (i.e., balanced usage), overly-used

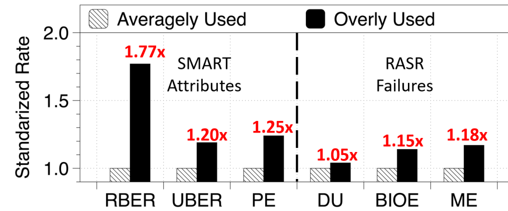


Figure 5: Comparison of overly used SSDs and averagely used SSDs. *This figure shows overly used SSDs exhibit more device level errors and RASR failures compared with averagely used SSDs. RBER: raw bit error rate; UBER: uncorrectable bit error rate; PE: program error count; DU: Drive Unfound; BIOE: Buffer IO Error; ME: Media Error.*

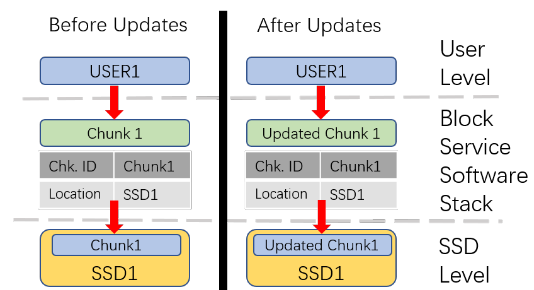


Figure 6: The data path of an update operation (original).

SSDs may exhibit more device-level errors and potentially lead to more RASR failures. To verify this hypothesis, we quantitatively measure such difference based on our dataset. We use the classic 80/20 rule to group the SSDs. The SSDs with top 20% usage within the Block Service are labeled as overly-used and the rest are labeled as averagely-used. As shown in Figure 5, overly-used SSDs have noticeably higher numbers of device errors including RBER (1.77X), UBER (1.20X) and PE (1.25X). Moreover, they tend to incur more RASR failures including Drive Unfound (1.05X), Buffer IO Error (1.15X), and Media Error (1.18X). This result suggests that load balancing is indeed important.

5.2 Root Causes of Usage Imbalance

After looking into the design of software stacks of the three cloud services, we identify two major factors for the unbalanced usage of SSDs in the Block service: the update policy and the user I/O patterns.

Figure 6 shows the simplified update policy in the Block service (for clarity, irrelevant details such as sharding and replication are omitted). The Block service offers users the storage capacity at the granularity of chunks. The left part of the figure shows that USER1 subscribes one chunk ("Chunk1") from the service. The software stack of the Block service maintains a mapping table from the chunk to a fixed SSD (i.e., storing "Chunk1" on"SSD1").

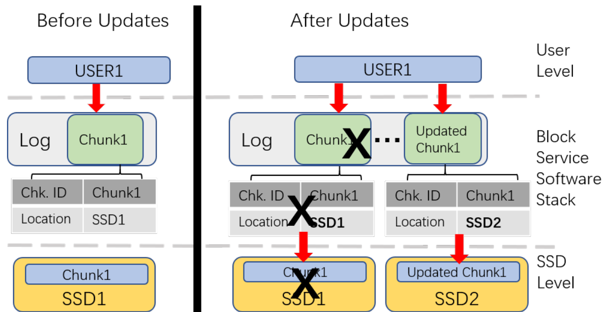


Figure 7: The data path of an update operation (optimized).

Upon an update operation, shown in the right part of Figure 6, the software stack queries the mapping table and writes the updated chunk to the *same* SSD (i.e., "Updated Chunk1" on "SSD1"). In other words, in the Block service performs *in-place* updates, i.e., updates are always flushed to the initially-allocated SSDs.

In addition, we find that the Block service receives a diverse set of I/O requests from different users. Some users generate many update operations while others do not. This diversity and the in-place update policy lead to the unbalanced usage of SSDs under the Block service.

Unlike Block Service, the other two cloud services do not cause severe usage imbalance because they have a different update policy or I/O pattern. Particularly, the NoSQL service merges small updates together and always generates a new chunk for the updated data, which can be mapped to a different SSD. In the Big Data service, reading and adding new data are consistently much more frequent than updating existing ones. Therefore, SSDs under both services have a relatively balanced usage.

5.3 Mitigating Usage Imbalance

To address the usage imbalance issue, we optimize the software stack of the Block service by adding a shared append-only log, similar to LFS [36].

Figure 7 shows the update operation after the optimization. Similar to Figure 6, USER1 subscribes a chunk from the Block Service. Unlike the original design, the chunk is now maintained in a log which appends the latest update to its end. Upon receiving an update, the software stack invalidates the previous chunk (marked with an "X"), appends the update to the log, and changes the mapping table to map the updated chunk to a new SSD ("SSD2"). The outdated chunk will be invalidated for garbage collection. This log-based design mitigates the usage variance among SSDs, as each updated chunk will be allocated to a different SSD based on the wear among available drives. Note that, in certain cases, the updated chunk may still mapped to the original SSD if the original one happens to be the most appropriate candidate.

After applying the optimized design on a subset of our target systems for 7 months, we observe that the coefficient of variance (CV) of host read/write under the Block service reduces significantly (i.e., from 24.9% to 5.2% among all SSDs under the same service). The log-structured design also provides other benefits for our target systems (e.g., better support for snapshots), which are beyond the scope of this paper.

Fix	Percentage	Root Cause
Rebooting	11.9%	Transient
Mount Options Check	0.4%	Human Mistake
FSCK	16.5%	Undetermined
Data Check	6.0%	Undetermined
Slot Check	20.1%	Human Mistake
Replacing Cable	13.9%	Faulty Cable
Replacing SSD	31.2%	Failed Device

Table 11: Working fixes of RASR failures. *The first column shows working fixes of RASR failures. The 2nd column lists the percentage of RASR failures repaired by each fix. The 3rd column lists the corresponding root cause derived from the working fix.*

6 Lessons & Actions for System Admins

To help system administrators with better fix strategy, we need better understanding of the root causes of RASR failures. While the repair log of a RASR failure does not explicitly state the root cause, we can infer the potential root cause based on the successful fix in the log. For example, if a failure can only be fixed by replacing the SSD, it is likely that the root cause is a failed SSD. Table 11 shows all the seven fixes deployed in the repairing procedure of RASR failures, the percentage of RASR failures being successfully repaired by each fix, and the potential root causes.

We observe that not all RASR failures are caused by failed SSDs. There are two main non-SSD causes for RASR failures: (1) *human mistakes* contribute 20.5% of RASR failures, including plugging the device to the wrong slot ("Slot Check") and incorrect configuration ("Mount Options Check"); and (2) *faulty interconnections* fixed by replacing cable, which is outside of SSD, account for 13.9% of RASR failures. Note that, although "Replacing SSD" accounts for the most (31.2%) of RASR failures, we still leave it as the last resort in the fix strategy due to the high cost of the devices and labors. Hence, we are interested in whether faulty interconnections and human mistakes can be quickly diagnosed or largely avoided.

6.1 Faulty Interconnection

Faulty interconnection is a well-known issue in large-scale storage systems [28]. To fix the RASR failures (Drive Unfound) caused by faulty interconnection, replacing the cable between SSD and host is an effective method. However, our symptom-based repairing procedure (shown in Table 6) lists replacing cable as the third step to try out if a Drive Unfound

Fix	Heavy Group	Light Group
Rebooting	4.4%	25.0%
Slot Check	7.6%	35.7%
Repl. Cable	70.6%	24.2%
Repl. SSD	17.4%	15.1%

Table 12: Success rates of fixes in two SSD groups. *This table shows the success rate of each fix for the "Drive Unfound" failures in two SSD groups classified by the indicator.*

failure occurs. This incentivizes us to quest for good indicators of faulty interconnection. If successful, administrators can directly replace cable instead of trying the first two failed attempts – significantly improving the repairing procedures of Drive Unfound (a major source of RASR failures).

6.1.1 Identifying Potential Indicators

To find a suitable indicator for faulty interconnection, we study the correlation between five representative device errors (i.e., Ultra-DMA CRC (UCRC), RBER, Uncorrectable Errors, Program Errors, and End-to-End Errors) and faulty interconnection by using Spearman Rank Correlation Coefficient [12]. The result shows that only the UCRC has strong correlation with faulty interconnection. This generally indicates that the more UCRC errors an SSD has, the more likely a Drive Unfound failure is caused by faulty interconnection. Therefore, we select the number of UCRC errors for designing the indicator.

6.1.2 Refining the Indicator

Since UCRC errors may also occasionally caused by transient factors (e.g., voltage spike), it is necessary to set a proper threshold for indicating faulty interconnection. To this end, we apply a set of classic statistics methods (e.g., Kolmogorov-Smirnov Test [13]) to analyze the UCRC errors and derive the optimal threshold. We find that the distribution of UCRC errors follows the 80/20 rule (i.e., Pareto Law [13]). So if the accumulation of UCRC errors on an SSD is in the top 20% among all drives, we assign the SSD to the "Heavy" group. Our analysis shows that 17 is the best threshold for our systems. In other words, when an SSD has 17 or more UCRC errors, it is a strong indication of a faulty interconnection in the target systems. Note that the threshold can be re-calculated and updated periodically for the change of systems and environment (e.g., aging of SSDs, workload changes). We leave the sensitivity study of the threshold as future work.

6.1.3 Verifying the Indicator

We further use our existing dataset to verify the effectiveness of the UCRC-based indicator. Specifically, we first divide all SSDs into two groups based on the threshold: "Heavy" (above or equal to the threshold) and "Light" (below the threshold). Then, we calculate the successful rate of each fix candidate for the "Drive Unfound" failures in each group.

Table 12 demonstrates that our indicator for faulty interconnection would be very effective for improving the repairing procedure of Drive Unfound failures. Most (i.e., 70.6%) SSDs in the "Heavy" group have been successfully repaired by replacing cable. On the contrary, only 12.0% of SSDs in the group are fixed by the first two candidates (i.e., node rebooting and slot check). This result suggests that, it can significantly improve the successful rate of the first attempt if we directly replace cables for the drives that are severely affected with UCRC errors. As for the "Light" group of SSDs, whose root causes are not identified as faulty interconnection by our indicator, the successful rate of replacing cable is similar to the first two fix candidates. This shows that the existing repair procedure is good for "Light" group of SSDs.

6.1.4 Benefits of Using Indicator

We have applied the UCRC-based indicator to our target systems. With the new repairing procedure for Drive Unfound failures, if the number of UCRC errors in SSD is higher than the threshold, on-site engineers will start with replacing cable first.

One might think that rebooting is simple and it should be the first attempt no matter what the root cause is. However, the side effect of rebooting can be notable and cascading in large-scale production systems. For example, a node may hang at BIOS during reboot if the system drive is inaccessible due to a faulty cable, which may further trigger large data transfer in a 3-replica system. Therefore, when the root cause is likely to be a faulty cable (i.e., the heavy group), we use cable replacement first.

Based on the feedback of the on-site engineers on the new 89 cases of Drive Unfound (not included in our dataset), the indicator helps them reduce the repairing time by 21.1%, because of the saving on time that would have been spent on the first two unsuccessful attempts (i.e., node rebooting and slot check).

6.2 Human Mistakes & Solution

As shown in Table 11, human mistake is another major source of RASR failures. Particularly, plugging the device to the wrong slot (i.e., fixed by "Slot Check") accounts for 20.1% RASR failures. Although it may be part of human nature to err, we believe such mistakes should be avoided.

To address the issue, we design an approach called One Interface One Purpose (OIOP) for our latest and future deployment, where SSDs serve for different purposes use different hardware interfaces. Table 13 lists the interface for each type of SSD functionality in our target systems. We use U.2/M.2 interface for system drives as our motherboard usually has 1 or 2 such sockets. The NVMe interface is used for temporary storage and buffering as these SSDs require high bandwidth and low latency. The SATA interface is used for persistent storage for compatibility concerns (i.e., re-using current SSDs on new racks). With such an OIOP design, the

SSD Functionality	SSD Interface
System drive	U.2/M.2
Temporary storage	NVMe
Buffering writes	NVMe
Persistent storage	SATA

Table 13: Mapping between SSD functionality and interface.

SSDs and slots for different purposes are easily differentiable by system administrators. Note that these interfaces are unlikely to be transitional as each interface has its unique market/purpose (e.g., U.2/M.2 for embedded, NVMe for high-performance).

Although simple, the OIOP design has effectively reduced the RASR failures caused by human mistakes in practice. In the 6-month deployment of an OIOP storage system with about 100K SSDs, we only observe 3 RASR failures caused by plugging a device to a wrong slot. In stark contrast, we observe an average of 47 such cases on comparable size of current storage systems without OIOP.

Besides OIOP, another possible solution is to use a status light to differentiate the functionalities of drives. Status light has been used for indicating drive status in RAID systems [40], and it can be applied to motherboards without multiple interfaces. However, adding status lights requires support from hardware manufacturers.

7 Related Works

Our work is mainly related to the following three lines of research studies: (1) reliability of SSDs and SSD-based storage systems, (2) reliability of HDD-based storage system, and (3) large-scale failure studies.

Great efforts have been made on analyzing the reliability of SSDs and SSD-based storage systems [32, 33, 38, 43, 44, 45]. For example, Schroeder et al. [38] conduct a large-scale field study covering millions of drive days and analyze a wide range of device characteristics and errors (especially RBER and UBER) as well as their correlation. Meza et al. [32] study flash memory failures in the field as well as their correlation with other factors (e.g., data written from OS). Narayanan et al. [33] analyze the correlation between failed SSDs and other factors (e.g., hardware utilization). Our work is different in a number of ways. First, we focus on RASR failures, which have not been studied before. Second, our study covers system-level failure symptoms, repair procedures, root causes, as well as the casual relations among events. Third, we design and validate a set of simple yet effective solutions. Therefore, we believe our work is complementary to the existing efforts.

Research efforts on HDD-based storage stack are also abundant [7, 8, 28, 35]. For example, Jiang et al. [28] study the logs from around 40K storage systems and discover several findings including the significant contribution of physical components and protocol stacks in failures, the "bursty" failure pattern, and the benefit of using redundant interconnection. Bairavasundaram et al. [7] analyze over 1.5 million hard drives and find out the severity differences of data corruption among enterprise and nearline disks, the spatial and temporal locality of checksum mismatches, and the correlation of data corruption across different disks. Based on the same dataset, Bairavasundaram et al. [8] also analyze factors that contribute to the latent sector errors along with the trends and further explore possible remedies towards building a more robust storage subsystem. However, due to the difference in both hardware design and software support, their findings may not be directly applicable to SSD-based storage systems.

In addition, our work is closely related to two groups of studies on large-scale failures. The first group focuses on using failure reports (e.g., news and descriptive records) to understand failures in modern storage systems [21, 22, 23, 43]. For example, Gunawi et al. [23] collect around 100 hardware fail-slow reports across multiple large-scale deployments from several institutions and study the behaviors, root causes and lessons for dianosis of fail-slow failures. In the second group of studies, researchers have made efforts on diagnosing and detecting failures from software perspective. For instance, Huang et al. [25, 26] analyze the production systems deployed at Microsoft and discover a key feature of the gray failure, differential observability, which leads them to build a fast detection tool. Regarding the first group, our work is different as we use multiple log sources (e.g. SMART logs, kernel logs) to conduct quantitative analysis and further derive the causal relationships of the failures. Compared with the second group, our work targets the various aspects of system reliability maintenance, including not only the software but also the hardware and administration.

8 Conclusions

We study the characteristics of RASR failures in large-scale storage systems in this paper. Our study reveals the distribution, symptoms, and causes of RASR failures. Moreover, we derive several lessons on system reliability, including the passive heating phenomenon, the usage imbalance, human mistakes, etc. In addition, we design and validate a set of simple yet effective methods to address the issues observed. We believe our findings and solutions would be beneficial to the community, and could facilitate building highly-reliable SSD-based storage systems.

Acknowledgments

We thank the anonymous reviewers and Mahesh Balakrishnan (our shepherd) for their insightful feedback. We also thank Yong Wang, Qingda Lu, Cheng He in Alibaba for the invaluable discussion.

References

[1] Amazon elastic block store, 2018. https://aws.amazon.com/ebs/.

[2] fstrim(8) - linux man page, 2018. https://linux.die.net/man/8/fstrim.

[3] Google cloud datastore, 2018. https://cloud.google.com/datastore/.

[4] Self-monitoring, analysis and reporting technology (s.m.a.r.t.) attributes, 2018. https://en.wikipedia.org/wiki/S.M.A.R.T.

[5] AMVROSIADIS, G., BROWN, A. D., AND GOEL, A. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).

[6] ANDERSEN, D. G., AND SWANSON, S. Rethinking Flash in the Data Center. *IEEE Micro 30*, 4 (2010), 52–54.

[7] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS) 4*, 3 (2008), 8:1–8:28.

[8] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proceedings of the 2017 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2007).

[9] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of the 2012 Design, Automation Test in Europe Conference Exhibition (DATE)* (2012).

[10] CAI, Y., LUO, Y., GHOSE, S., AND MUTLU, O. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015).

[11] CHO, S. Nand reliability improvement with controller assisted algorithms in ssd. In *Proceedings of the 2013 Flash Memory Summit* (2013).

[12] CORDER, G., AND FOREMAN, D. *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, 2014.

[13] DANIEL, W. *Applied nonparametric statistics*. PWS-Kent Publ., 1990.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)* (2010).

[15] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing space amplification in rocksdb. In *Proceedings of the 8th biennial Conference on Innovative Data Systems Research (CIDR)* (2017).

[16] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[17] GARRAGHAN, P., TOWNEND, P., AND XU, J. An empirical failure-analysis of a large-scale cloud computing environment. In *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering* (2014).

[18] GARULLI, N., BONI, A., CASELLI, M., MAGNANINI, A., AND TONELLI, M. A low power temperature sensor for iot applications in cmos 65nm technology. In *Proceedings of the 7th IEEE International Conference on Consumer Electronics - Berlin (ICCE-Berlin)* (2017).

[19] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[20] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2009).

[21] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)* (2014).

[22] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)* (2016).

[23] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[24] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D. R., CHIEN, A. A., AND GUNAWI, H. S. The Tail at Store - A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)* (2016).

[25] HUANG, P., GUO, C., LORCH, J. R., ZHOU, L., AND DANG, Y. Capturing and enhancing in situ system observability for failure detection. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[26] HUANG, R., GUO, C., ZHOU, L., LORCH, J., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[27] JEDEC. *Solid-State Drive (SSD) Requirements and Endurance Test Method.*, Sept. 2010.

[28] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures? *ACM Transactions on Storage 4*, 3 (2008), 1–25.

[29] LIANG, Y., ZHANG, Y., SIVASUBRAMANIAM, A., JETTE, M., AND SAHOO, R. Bluegene/l failure analysis and prediction models. In *Proceedings of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2006).

[30] LUO, Y., GHOSE, S., CAI, Y., HARATSCH, E. F., AND MUTLU, O. Heatwatch: Improving 3d nand flash memory device reliability by exploiting self-recovery and temperature awareness. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018).

[31] MAHDISOLTANI, F., STEFANOVICI, I., AND SCHROEDER, B. Proactive error prediction to improve storage system reliability. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017).

[32] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2015).

[33] NARAYANAN, I., WANG, D., JEON, M., SHARMA, B., CAULFIELD, L., SIVASUBRAMANIAM, A., CUTLER, B., LIU, J., KHESSIB, B. M., AND VAID, K. SSD Failures in Datacenters - What? When? and Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR)* (2016).

[34] PAPANDREOU, N., PARNELL, T., POZIDIS, H., MITTELHOLZER, T., ELEFTHERIOU, E., CAMP, C., GRIFFIN, T., TRESSLER, G., AND WALLS, A. Using adaptive read voltage thresholds to enhance the reliability of mlc nand flash memory systems. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI)* (2014).

[35] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (2007).

[36] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[37] SCHROEDER, B., AND GIBSON, G. A. Disk Failures in the Real World - What Does an MTTF of 1, 000, 000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (2007).

[38] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash Reliability Production - The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)* (2016).

[39] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010).

[40] SUN MICROSYSTEMS. *Sun StorEdge 3000 Family Installation, Operation, and Service Manual.*

[41] 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 *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)* (2017).

[42] VISHWANATH, K., AND NAGAPPAN, N. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)* (2010).

[43] XU, E., ZHENG, M., QIN, F., WU, J., AND XU, Y. Understanding SSD reliability in large-scale cloud systems. In *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)* (2018).

[44] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of ssds under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (2013).

[45] ZHENG, M., TUCEK, J., QIN, F., LILLIBRIDGE, M., ZHAO, B. W., AND YANG, E. S. Reliability analysis of ssds under power fault. *ACM Transactions on Computer Systems (TOCS) 34*, 4 (2017), 10:1–10:28.

Who's Afraid of Uncorrectable Bit Errors?
Online Recovery of Flash Errors with Distributed Redundancy

Amy Tai
Princeton University and VMware Research

Andrew Kryczka
Facebook Inc.

Shobhit O. Kanaujia
Facebook Inc.

Kyle Jamieson
Princeton University

Michael J. Freedman
Princeton University

Asaf Cidon
Columbia University

Abstract

Due to its high performance and decreasing cost per bit, flash storage is the main storage medium in datacenters for hot data. However, flash endurance is a perpetual problem, and due to technology trends, subsequent generations of flash devices exhibit progressively shorter lifetimes before they experience uncorrectable bit errors. In this paper, we present an approach for addressing the flash lifetime problem by allowing devices to operate at much higher bit error rates. We present DIRECT, a set of techniques that harnesses distributed-level redundancy to enable the adoption of new generations of denser and less reliable flash storage technologies. DIRECT does so by using an end-to-end approach to increase the reliability of distributed storage systems.

We implemented DIRECT on two real-world storage systems: ZippyDB, a distributed key-value store in production at Facebook that is backed by and supports transactions on top of RocksDB, and HDFS, a distributed file system. When tested on production traces at Facebook, DIRECT reduces application-visible error rates in ZippyDB by more than 100× and recovery time by more than 10,000×. DIRECT also allows HDFS to tolerate a 10,000–100,000× higher bit error rate without experiencing application-visible errors. By significantly increasing the availability of distributed storage systems in the face of bit errors, DIRECT helps extend flash lifetimes.

1 Introduction

Flash has become the dominant storage medium for hot data in datacenters [64, 72], since it offers significantly lower latency and higher throughput than hard disks. Many storage systems are built atop flash, including databases [6, 11, 15, 44], caches [5, 36, 57, 58, 78], and file systems [48, 67].

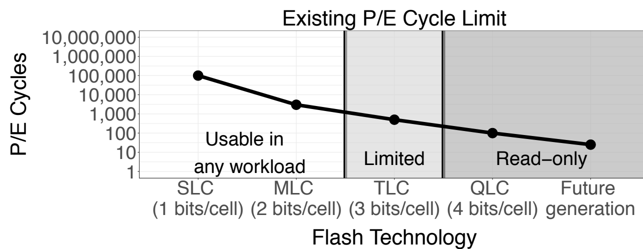
However, a perennial problem of flash is its limited endurance, or how long it can reliably correct raw bit errors. As device writes are the main contributor to flash wear, its lifetime is measured in the number of writes or program-erase (P/E) cycles the device can tolerate before exceeding an uncorrectable bit error threshold. Uncorrectable bit errors are errors that are exposed externally and occur when there are too many raw bit errors for the device to correct.

In hyper-scale datacenters, operators constantly seek to reduce flash wear by limiting flash writes [21, 64]. At Facebook, for example, a dedicated team monitors application writes to ensure they do not prematurely exceed manufacturer-defined device lifetimes. Even worse, each subsequent flash generation tolerates a smaller number of writes before reaching end-of-life (see Figure 1a) [42]. Further, given the scaling challenges of DRAM [49, 56] and the increasing cost gap between DRAM and flash [2, 37, 38], many operators are migrating services from DRAM to flash [7, 37], increasing the pressure on flash lifetime.

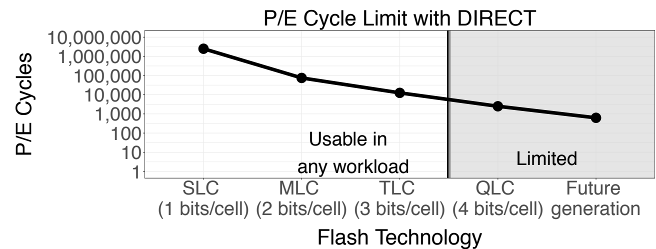
There is a variety of work that attempts to extend flash lifetime by delaying the onset of bit errors [6, 12, 30, 36, 47, 59, 61, 62, 63, 77, 82, 83]. This paper takes a contrarian approach. We observe that flash endurance can be extended by *allowing* devices to go beyond their advertised uncorrectable bit error rate (UBER) and embracing the use of flash disks that exhibit much higher error rates; Google recently released a whitepaper suggesting a similar approach [28]. We can do so without sacrificing durability because datacenter storage systems replicate data on remote servers, and this redundancy can correct bit error rates orders of magnitude beyond the hardware error correction mechanisms implemented on the device. However, the challenge with higher flash error rates is maintaining availability and correctness.

We introduce Distributed error Isolation and RECovery Techniques (DIRECT), which is a set of three simple general-purpose techniques that, when implemented, enable distributed storage systems to achieve high availability and correctness in the face of uncorrectable bit errors:

1. **Minimize data error amplification.** DIRECT detects errors using existing error detection mechanisms (e.g., checksums) and recovers data from remote servers at the smallest possible granularity.
2. **Minimize metadata error amplification.** A corruption in local metadata (e.g., database index), often requires a large amount of data to be re-replicated. DIRECT avoids



(a) Existing hardware-based error correction.



(b) Augmenting existing error correction with DIRECT.

Figure 1: For each generation of flash bit density, the average number of P/E cycles after which the uncorrectable bit error rate falls below the manufacturer specified level (10^{-15}). Beyond MLC, the number of flash writes the application can issue is limited [29]. With current hardware-based error correction, QLC technology and beyond can only be used for applications that are effectively read-only [8, 22, 76]. DIRECT enables the adoption of denser flash technologies by handling errors in the distributed storage application. We use the model from §3 to compute the UBER tolerated by DIRECT, while the UBER to P/E conversion was derived from data in a Google study [72].

this by adding redundancy locally to local metadata.

3. **Ensure safe recovery semantics** by treating recovery operations as write operations. DIRECT serializes recovery operations on corrupted data against concurrent operations with respect to the system's consistency guarantees.

The difficulty of implementing DIRECT depends on two properties of the underlying storage system. The first property is whether the system is physically or logically replicated. Physically-replicated systems replicate *data blocks* between servers, while logically-replicated systems replicate the *commands* (e.g., write, update, delete). In physically-replicated systems, a certain object is stored in the same block or file on another server and therefore can be recovered efficiently by simply re-replicating the remote data block. This does not work for logically-replicated systems, where physical blocks are not identical across replicas. The second property is whether the data store supports versioning. In systems with versioning, we need to guarantee the recovered object does not override a more up-to-date version.

We demonstrate how to generalize DIRECT techniques by implementing them in two popular systems that are representative of two different classes of storage systems: (1) the Hadoop Distributed File System (HDFS), which is a physically-replicated storage system without versioning, and (2) ZippyDB, a distributed system that implements logical replication and transactions on top of RocksDB, a popular key-value store that supports key versioning. Objects in HDFS are physically-replicated, so it is straightforward for DIRECT to find the corrupt object in another replica and recover it at a high granularity (§4.1). On the other hand, recovery is challenging in ZippyDB since the corrupted region of one replica is stored in a different location on another replica, so the recovered key-value pairs might not have consistent versions ZippyDB (§4.2).

DIRECT Limitations and Lessons Learned. We chose to implement DIRECT by retrofitting existing datacenter storage applications, rather than as a general-purpose application library. The latter design would be particularly challenging since DIRECT depends on application-specific details such as file layout and recovery semantics. Note that the storage systems we retrofitted (HDFS and RocksDB) and their relatives serve as the base layer for many storage services and databases (e.g., MyRocks, Ozone, HBase, Cassandra). Furthermore, we learned that to implement the first and third DIRECT techniques, storage systems must have a key requirement: we must be able to infer logical objects from the physical location (on the application's file format) of the bit error. In §6 we discuss PostgreSQL, which does not satisfy this requirement, and therefore is difficult to retrofit with DIRECT.

DIRECT leads to significant increases in device lifetime, since systems can maintain the same probability of application-visible errors (durability) at much higher device UBERs. In Figure 1b, we estimate the number of P/E cycles gained with DIRECT using an empirical UBER vs P/E cycle comparisons in a Google study [72]. Depending on workload parameters and hardware specifications, DIRECT can increase the lifetime of devices by 10-100×. This allows datacenter operators to replace flash devices less often and adopt lower cost-per-bit flash technologies that have lower endurance. DIRECT also provides the opportunity to rethink the design of existing flash-based storage systems, by removing the assumption that the device fixes all corruption errors. Furthermore, while this paper focuses on flash, DIRECT's principles also apply in other storage mediums, including NVM, hard disks, and DRAM.

In summary, this paper makes several contributions:

1. We observe flash lifetime can be extended by allowing devices to operate at much higher bit error rates.
2. We propose DIRECT, general software techniques that enable storage systems to maintain performance and high availabily despite high hardware bit error rates.
3. We design and implement DIRECT in two storage systems, HDFS and ZippyDB, that are representative of physical and logical replication, respectively. Applying DIRECT results in significant end-to-end availability improvements: it enables HDFS to tolerate bit error rates that are 10,000×-100,000× greater, reduces application-

visible error rates in ZippyDB by more than 100×, and speeds up recovery time in ZippyDB by 10,000×.

2 Motivation

What Limits Flash Endurance? Flash chips are composed of memory cells, each of which stores an analog voltage value. The flash controller reads the value stored in a certain memory cell by sensing the voltage level of the cell and applying quantization to determine the discrete value in bits. The more bits stored in a cell, the narrower the voltage range that maps to each discrete bit, so more precise voltage sensing is required to get a correct read. A primary way to reduce cost per bit is to increase the number of bits per cell, which means that even small voltage perturbations can result in a misread.

Multiple factors cause voltage drift in a flash cell. The dominant source, especially in datacenter settings where most data is "hot," is the program-erase (P/E) cycle, which involves applying a large high voltage to the cell in order to drain its stored charge, thus wearing the insulating layer in the flash cell [30]. This increases the voltage drift of subsequent values in the cell, which gradually leads to bit errors.

3D NAND is a recent technology that has been adopted for further increasing flash density by stacking cells vertically. While 3D NAND relaxes physical limitations of 2D NAND (traditional flash) by enabling vertical stacking, 3D NAND inherits the reliability problems of 2D NAND and further exacerbates them, since a cell in 3D NAND has more adjacent (vertical) neighbors. For example, voltage retention is worse, because voltage can now leak in three dimensions [54, 63, 65]. Similarly, disturb errors that occur when adjacent cells are read or programmed are also exacerbated [50, 75].

Existing Hardware Reliability Mechanisms and Limitations. To correct bit errors, flash devices use error correcting codes (ECC), which are implemented in hardware. After the ECC pass, there could still be incorrect bits on the page. To address this, SSDs also employ internal RAID across the dies of a flash device [16, 19]. After applying coding and RAID within the device, there will remain a certain rate of *uncorrectable bit errors* (UBER). Together, ECC and internal RAID mechanisms can drive the error rates of SSDs from the raw bit error rate of around 10^{-6} down to the 10^{-17} to 10^{-20} UBER range typical of enterprise SSDs [14]. "Commodity" SSD devices typically guarantee an UBER of 10^{-15}.

While it is possible to create stronger ECC engines, the higher the corrective power of the ECC, the more costly the device [4, 10]. Furthermore, the level of internal RAID striping is constant across generations, because the number of dies inside a flash device remains constant. This means that the corrective power of RAID is fixed.

Similarly, while RAID across devices [53, 69, 74] can add redundancy, a main design goal of DIRECT is to avoid adding unnecessary overhead. We avoid turning to RAID because it is inflexible since its recovery power is fixed at deployment time, and, more importantly, it imposes storage and write overheads, in particular generating additional flash writes that further reduce endurance.

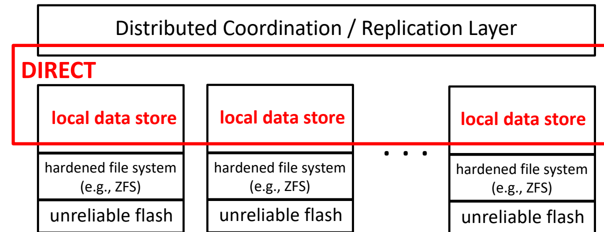


Figure 2: DIRECT fixes errors in the local data store, sometimes requiring interaction with the distributed coordination layer.

Implications of Limited Flash Endurance. Flash technology has already reached the point where its endurance is inhibiting adoption and operation in various datacenter use cases. For example, QLC was recently introduced as the next generation flash cell technology. However, in the worst case, it can only tolerate ~150 P/E cycles [8, 22, 76], so it can only be used for read-heavy use cases, e.g., a 2 TB QLC drive with a lifetime of 150 P/E cycles can only write at a rate of 2 MB/s or less in order to preserve its advertised lifetime of 5 years. In the best case, some QLC devices can tolerate ~1000 P/E cycles for completely sequential write workloads, due to an internal SLC cache [8]. But since datacenter applications like databases and analytics that deal with hot data typically need to update objects frequently, the adoption of QLC has been more limited and is the reason that Facebook has avoided QLC flash. Subsequent cell technology generations will suffer from even greater problems.

Operational issues also often dictate a device's usage lifetime. Flash is typically only used for its advertised lifetime to simplify operational complexity [72]. Further, in a hyperscale datacenter where it is common to source devices from multiple vendors, the most conservative estimate of device lifetime across vendors is typically chosen as the lifetime for a fleet of flash devices, so that the entire fleet can be installed and removed together. If the distributed storage layer could tolerate much higher device error rates, then datacenter operators would no longer have to make conservative and wasteful estimates about entire fleets of flash devices.

Finally, because of the increase in DRAM prices due to its scaling challenges and tight supply [2, 38, 49, 56], datacenter operators are migrating services from DRAM to flash [7, 37]. This means flash will be responsible for many more workloads, further exacerbating its endurance problem.

3 DIRECT Design

DIRECT is a set of techniques that enables a distributed storage system to maintain high availability and correctness in the face of high UBER.

We define a distributed storage system as a set of many local stores coupled with a distributed protocol layer that replicates data and coordinates between the local stores. Figure 2

shows an ideal storage stack that runs on unreliable flash (flash that exposes high UBERs). Note that there is existing work on how to make local file systems tolerate corruption errors (we survey these in §6), so our efforts in this paper focus on hardening the application-level storage system.

We observe that redundancy already exists in distributed storage systems in the form of distributed replication [27, 32, 40], which maintains multiple remote copies of each piece of data, or distributed erasure coding [45, 71, 80], which maintains remote parity bits for each piece of data. However, many systems do not systematically use this redundancy to recover individual bit errors [39], even though it can significantly boost resilience to bit errors. We focus on using distributed replicas to correct bit errors, but the DIRECT principles presented in the paper also apply to systems that use erasure coding; for example, error amplification — the number of bits required to recover an error — can be reduced in erasure coding schemes by reducing the size of a stripe.

Distributed Redundancy. Consider the following example of a physically replicated storage system, such as HDFS. Suppose the minimum unit of recovery is a data block[1], which is replicated in each of three data stores. If the block has size B, and the uncorrectable bit error rate (UBER) is E, then the expected number of errors in the block will be $B \cdot E$. Since the block is replicated across R different servers, the only way that the storage system would encounter an application-observable read error is when at least one error exists in *every* copy of the block. Therefore, the probability of an application-level read error can be expressed as:

$$\mathbb{P}[error] = (1-(1-E)^B)^R \approx (B \cdot E)^R$$

where we assume $B \cdot E << 1$ and use a Taylor series approximation.

Then, for an UBER of $E = 10^{-15}$, a block size of $B = 128\ MB$ (typical of distributed file systems), and a replication factor of $R = 3$, the probability of a read error is 10^{-18} (files are measured in bytes, while UBER is in bits).

However, with relatively large blocks, the probability of encountering at least one error in all block replicas quickly increases as UBER increases. For example, for an UBER of $E = 10^{-10}$, the expected number of errors in a single block will be $B \cdot E = 0.1$ for 128 MB blocks (Table 1). Then in this case $\mathbb{P}[error] \approx 0.001$. We observe that reducing $B \cdot E$, by reducing B, will dramatically reduce the probability of error.

Minimizing Error Amplification of Data Blocks. DIRECT captures this intuition with the notion of *error amplification* (B in the previous example), or the number of bytes required to recover a bit error. DIRECT observes that *the lower the error amplification, the lower the probability of error and the faster recovery can occur*. This similarly implies a shorter period of time spent in degraded durability

[1]Note that in HDFS while errors can be detected using checksums at a smaller granularity than the block size, actual recovery and replication is conducted at the granularity of a block.

UBER	*Probability of Application-Observable Error* Block Recovery	Chunk Recovery
10^{-10}	$1 \cdot 10^{-3}$	$3 \cdot 10^{-10}$
10^{-15}	$1 \cdot 10^{-18}$	$1 \cdot 10^{-28}$

Table 1: Probability of application-observable error comparing block-by-block recovery to chunk-by-chunk recovery, with an UBER of 10^{-10}, and 10^{-15}. Finer granularity recovery provides significantly higher protection against corruptions.

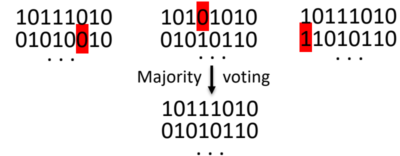


Figure 3: Even if the same chunk is corrupted on all replicas, bit-by-bit majority voting can reconstruct the correct chunk, by taking the majority vote of each bit across all chunks.

and thus higher availability.

In the example above, suppose the system can recover data at a finer granularity, for example, at chunk size $C = 64$ KB. Then a read error would occur if all three replicas of the same *chunk* have at least one bit error. The revised probability of read error becomes:

$$\mathbb{P}[error] = 1-(1-(1-(1-E)^C)^R)^{\frac{B}{C}}$$

Assuming $E \cdot C << 1$, Taylor series approximation leads to $(1-(1-E)^C)^R) \approx (E \cdot C)^R$, and assuming this value is much smaller than $\frac{B}{C}$, the probability of an application-observable error when correcting chunk-by-chunk is:

$$\mathbb{P}[error] \approx (E \cdot C)^R \cdot \frac{B}{C}$$

When $C = 64$ KB and $E = 10^{-10}$, this probability is $3 \cdot 10^{-10}$, which is much lower than the probability when recovering at the block level (see Table 1).

We can further reduce B by using bit-by-bit majority voting, i.e., the recovered value of a bit in the chunk is the majority vote across the three chunk replicas (Figure 3). Bit-by-bit majority voting further reduces the application-observable error beyond chunk-by-chunk recovery, because the only way an application-observable error would occur is if an error occurs in the same bit across two chunks or more.

In a physically-replicated system like HDFS, minimizing error amplification is straightforward because corrupted blocks (and even bits) can be directly recovered from remote replicas. For a logically-replicated system like ZippyDB, however, blocks are not identical across replicas. This makes minimizing error amplification more challenging, since DIRECT cannot simply recover from a remote physical chunk. For example, bit-by-bit majority voting is not possible in ZippyDB, because the replicas do not store the same physical bits. For such systems, DIRECT must instead first isolate the region where the error might have occurred and then retrieve objects one-by-one from the other servers (see §4.2).

Minimizing Metadata Error Amplification. Error amplification can be even more severe if the error occurs in local metadata. For example, a corrupt index in a key-value store can prevent a data store from starting up, which can mean re-replication of hundreds of GBs of data. Thus even though the likelihood of errors in metadata is lower than in data blocks (metadata typically takes up less space than data), it still requires protection. Hence DIRECT either locally duplicates metadata or applies local software error correction.

Safe Recovery Semantics. DIRECT must also ensure recovery operations preserve the correctness of the distributed storage system, which might be dealing with concurrent write and read operations.

This is relatively straightforward in systems that do not support versioning or transactions, such as HDFS, since an object is up-to-date as soon as it is recovered from a remote replica. Systems like RocksDB which support versioning are more challenging, because if the system re-writes an object from a remote replica, recovery might overwrite a newer version with a stale version. In particular, the versions of the corrupted key-value pairs *are not known*, because (a) the corruption prevents the data from being read and (b) due to logical replication, the data's location does not provide information on its version. Hence to correctly recover corrupted key-value pairs, the system must locate some consistent (up-to-date) version of each pair. To do this, DIRECT forces recovery operations to go through a fault-tolerant log (for ZippyDB we use its existing Paxos log), which can provide correct ordering (§4.2.3).

DIRECT Techniques. To summarize, DIRECT includes the following techniques.

1. Systems must reduce error amplification of data objects and fix corruptions from remote replicas.
2. Systems must reduce local metadata error amplification, which is much higher than data error amplification.
3. Systems must ensure safe recovery semantics.

Note that the first and second techniques apply exclusively to the local data store and affect *performance*, while the third technique may require that the local data store interact with the distributed coordination layer to ensure *correctness*.

4 Implementing DIRECT

To demonstrate the use of the DIRECT approach, we integrate it into two systems: HDFS, a popular distributed file system, and ZippyDB, a distributed key-value store backed by RocksDB. The techniques used to implement DIRECT in HDFS can be applied to other physically replicated systems, such as GFS [40], Windows Azure Storage [31], and RAMCloud [66], which write objects into large immutable blocks that are replicated across several servers. Similarly, the techniques used to implement DIRECT in ZippyDB and RocksDB can be applied to other logically replicated systems, such as Cassandra [79], MongoDB [9], and CockroachDB [1]. In these systems a distributed coordination layer manages the replication of objects across different servers and uses versioning to execute transactions.

4.1 HDFS-DIRECT

4.1.1 HDFS Overview.

HDFS is a distributed file system that is designed for storing large files that are sequentially written and read. Files are divided into 128MB blocks, and HDFS replicates and reads at the block level.

There are three types of HDFS servers: NameNode, JournalNode, and DataNode. The NameNode and JournalNodes store cluster metadata by running a protocol similar to Multi-Paxos; we note that this protocol can tolerate bit errors by writing an additional entry per Paxos entry (for more information, see PAR [20]). DataNodes (the local data stores in Figure 2) store HDFS data blocks, and they respond to client requests to read blocks. If a client encounters errors while reading a block, it will continue trying other DataNodes from the offset of the error until it can read the entire block. After an error on a DataNode, the client will not try that node again. If there are no more DataNodes and the block is not fully read, the read fails and that block is considered missing.

Additionally, HDFS has a configurable background "block scanner" that periodically scans data blocks and reports corrupted blocks for re-replication. But the default scan interval is three weeks, and the scanner still recovers at the 128 MB block granularity. If there is a bit error in every replica of a block, then HDFS cannot recover the block.

4.1.2 Implementing DIRECT

Minimizing Error Amplification of Data Blocks. We leverage the observation that HDFS checksums every 512 bytes in each 128 MB data block. Corruptions thus can be narrowed down to a 512 byte chunk; verifying checksums adds no overhead, because by default HDFS will verify checksums during every block read. For streaming performance, the smallest-size buffer that is streamed during a data block read is 64 KB, so we actually repair 64 KB everytime there is a corruption. To mask corruption errors from clients, we repair a data block synchronously during a read. Under DIRECT, the full read (and recovery) protocol is the following.

Each 128 MB block in HDFS is replicated on three DataNodes, call them A,B,C. An HDFS read of a 128 MB block is routed to one of these DataNodes, say A. A will stream the block to the client in 64 KB chunks, verifying checksums before it sends a chunk. If there is a checksum error in a 64 KB chunk, then A will attempt to repair the chunk by requesting the 64 KB chunk from B. If the chunk sent by B also contains a corruption, then A will request the chunk from C.

If C *also* sends a corrupted chunk, then A will attempt to construct a correct version of the chunk through bit-by-bit majority voting: the value of the bit in the chunk is the majority vote across the three versions provided by A, B, and C. After reconstructing the chunk via majority voting (Figure 3), A will verify the checksums again; if the checksums fail, then the

read fails. Majority voting allows HDFS-DIRECT to tolerate on the order of $10^4 - 10^5$ times more bit errors than HDFS. In fact, as we show in Section 5.1, UBERs can be as high as 10^{-5} before majority voting failures are detectable in our experimental framework

Safe Recovery Semantics. Safety is straightforward in HDFS because data blocks are immutable once written, so there are never updates that will conflict with chunk recovery. Before a client does a block read, it first contacts the NameNode to get the DataNode IDs of all the DataNodes on which the block is replicated. In HDFS-DIRECT, when a client sends a block read request to a DataNode, it also sends this set of IDs. Because blocks are immutable and do not contain versions, these IDs are guaranteed to be correct replicas of the block, *if they exist*. It could be that a concurrent operation has deleted the block. In this case, if chunk recovery cannot find the block on another DataNode because it has been deleted, then it cannot perform recovery, so it will return the original checksum error to the client. This is correct, because there is no guarantee in HDFS that concurrent read operations should see the instantaneous deletion of a block.

Minimizing Metadata Error Amplification. Each server in HDFS has local metadata files that must be correct, otherwise it cannot start. These files include a VERSION file, as well as special files on the NameNode and JournalNode. Metadata files are not protected in HDFS, thus a single corruption will prevent the server from starting. DIRECT adds a standard CRC32 checksum at the beginning of each file and replicates the file twice so that there are three copies of the file on disk.

4.2 ZippyDB-DIRECT

4.2.1 ZippyDB Overview

We also implemented DIRECT on a logically replicated system, ZippyDB, a distributed key-value store used within Facebook that is backed by RocksDB (i.e., RocksDB is the local data store in Figure 2), which is a versioned key-value store.

ZippyDB runs on tens of thousands of flash-provisioned servers at Facebook, which makes it an ideal target for DIRECT. ZippyDB provides a replication layer on top of RocksDB. ZippyDB is logically separated into shards, and each shard is fully replicated at least three ways. Each shard has a primary replica as well as a number of secondary replicas, wherein each replica is backed by a separate RocksDB instance residing on some server. Each ZippyDB server contains hundreds of shards, including both primary and secondary replicas. Hence, each ZippyDB server actually contains a large number of separate RocksDB instances.

ZippyDB runs a Paxos-based protocol for shard operations to ensure consistency. The primary shard acts as the leader for the Paxos entry, and each shard also has a Paxos log to persist each Paxos entry. Writes are considered durable when they are committed by a quorum of shards, and write operations are applied to the local RocksDB store in the order that they are committed. A separate service is responsible for monitoring the primary and triggering Paxos role changes.

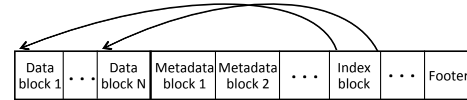


Figure 4: RocksDB SST file format. Index block entries point to keys within data blocks. Therefore, consecutive index entries form a key range. DIRECT modifies this file format by writing each metadata block at least twice in-line.

4.2.2 RocksDB Overview

RocksDB is a local key-value store based on a log-structured merge (LSM) tree [68]. RocksDB batches writes in-memory—each write receives a sequence number that enables key versioning—and flushes them into immutable files of sorted key-value pairs called sorted string table (SST) files. SST files are composed of individually checksummed blocks, each of which can be a data block or a metadata block. The metadata blocks include index blocks whose entries point to the keys at the start of each data block (see Figure 4) [13].

SST files are organized into levels. A key feature of RocksDB and other LSM tree-backed stores is background compaction, which periodically scans SST files and compacts them into lower levels, as well as performs garbage collection on deleted and overwritten keys.

4.2.3 Implementing DIRECT

ZippyDB has high error amplification since a single bit error can cause migration of terabytes of data: if a compaction encounters a corruption, an entire server, which typically has hundreds of gigabytes to terabytes of data, will shutdown and attempt to drain its RocksDB shards to another machine. Meanwhile, this sudden crash causes spikes in error rates and increases the load on other replicas while the server is recovering. To make matters worse, the new server could reside in a separate region, further delaying time to recovery.

Minimizing Error Amplification of Data Blocks. We observe that checksums in RocksDB are applied at the data block level, so a data block is the smallest recovery granularity. Data blocks are lists of key-value pairs, and key-value pairs are replicated at the ZippyDB layer. A corrupted data block can be recovered by fetching the pairs in the data block from another replica. However, this is challenging for two reasons.

First, compactions are non-deterministic in RocksDB and depend on a variety of factors, such as available disk space and how compaction threads are scheduled. Hence, *two replicas of the same RocksDB instance will have different SST files*, making it impossible to find an exact replica of the corrupted SST file and the corrupted data block, unlike in HDFS. Second, because the block is corrupted, it is impossible to know the exact key-value pairs that were stored in that block.

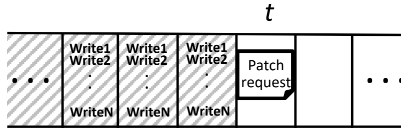


Figure 5: To serialize a patch properly, we add it as a request in the Paxos log. If the patch request is serialized at point t, then it must reflect all entries $t' < t$ (shaded). Furthermore, the patch request is not batched with any writes to ensure atomicity.

Therefore, not only do we not know what data to look for on the other replica, we also don't know where to find it.

Instead of repairing the exact keys that are lost, we rewrite a larger key range that covers the keys in the corrupted block. The key range is determined from index blocks, which are a type of metadata block in SST files that has an entry for the first key of each data block (Figure 4). Hence, consecutive index block entries form a key range which is guaranteed to contain the lost keys. Note that relying on these index entries requires that the index block, a metadata block, be error-free. See below for how we ensure the index block is uncorrupted.

Unfortunately, just knowing the key range is not enough: the existence of key versions in RocksDB and quorum replication in ZippyDB compounds the problem. In particular, a key must be recovered to a version greater than or equal to the lost key version, which could mean deleting it as key versions in RocksDB can be deletion markers. Additionally, if we naïvely fetch key versions from another replica, we may violate consistency.

Safe Recovery Semantics. To guide our recovery design, we introduce the following correctness requirement. Suppose we learn from the index entries that we must re-replicate key range $[a,b]$. This key range is requested from another replica, which assembles a set of fresh key-value pairs in $[a,b]$, which we call a patch.

Safety Requirement: *Immediately after patch insertion, the database must be in a state that reflects some prefix of the Paxos log. Furthermore, this prefix must include the Paxos entries that originally updated the corrupted data block.*

In other words, patch insertion must bring ZippyDB to some consistent state *after* the versions of the corrupted keys; otherwise, if the patch inserts prior versions of the keys, then the database will appear to go backwards.

Because the Paxos log serializes updates to ZippyDB, the cleanest way to find a prefix to recover up to is to serialize the patch insertion via the Paxos log. Then if patch insertion gets serialized as entry t in the log, the log prefix the patch must reflect is all Paxos entries $t' < t$, as shown in Figure 5. Serializing a patch at index t tells us exactly how to populate the patch. In particular, each key in the patch must be recovered to the largest entry $s < t$ such that s is the index of a Paxos entry that updates that key.

Furthermore, patch insertion must be atomic. Otherwise, it could be interleaved with updates to keys in the patch, which would violate the safety requirement, because then the version

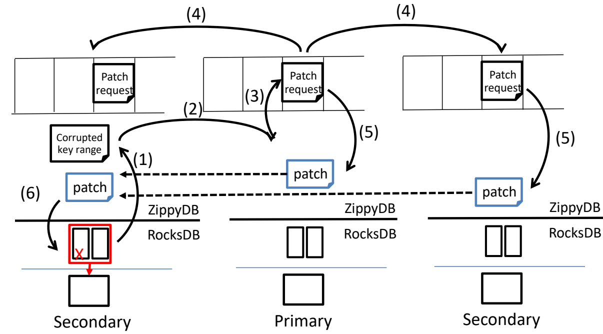


Figure 6: The process of recovering a corrupted RocksDB data block: (1) RocksDB compaction iterator determines the corrupted key range(s) based on the index blocks of the SST files and reports it to ZippyDB. (2) ZippyDB reports this error to the primary of that replica. (3) Primary shard adds patch request to Paxos log. (4) Paxos engine replicates the request to all replicas. (5) Each replica tries to process the patch request. If the processing shard is *not* the corrupted shard (which it knows because the patch request contains the shard ID of the corrupted shard), then it prepares a patch from its local RocksDB state and sends it to the corrupted shard. If the processing shard *is* the corrupted shard, then it waits for a patch from any replica. (6) Corrupted shard applies the fresh patch to its local RocksDB store.

of the key in the patch would not reflect a prefix of t. This is actually a subtle point because ZippyDB batches many writes into a single Paxos entry, as shown in Figure 5. If patch insertion is batched with other writes, then the patch will not reflect the writes that are in front of it in the batch. Hence, we force the patch insertion to be its own Paxos entry.

Minimizing Metadata Error Amplification. There are two flavors of metadata in RocksDB: metadata files and metadata blocks in SST files. Metadata files are only read during startup and then cached in memory. We can easily protect them with local replication, which adds a minimal space overhead (on the order of kilobytes per server). We protect metadata blocks by writing them several times in-line in the same SST file. In our implementation, we write each metadata block twice[2]. Protecting metadata enables us to isolate errors to a single data block, rather than invalidating an entire SST file.

As with the HDFS JournalNode, we can protect against errors in the ZippyDB Paxos log with an additional entry [20].

4.2.4 DIRECT Recovery in ZippyDB

ZippyDB-DIRECT triggers a recovery procedure when RocksDB encounters a corruption error during compaction. For user reads, ZippyDB does not synchronously recover corrupted blocks, unlike in HDFS. Instead, it returns the error to the client, which will retry on a different replica, and ZippyDB will then trigger a manual compaction involving the file of the corrupted data block.

Figure 6 depicts this process. Importantly, we do not release a compaction's output files until the recovery procedure

[2] For increased protection, metadata blocks can be locally replicated more than twice or protected with software error correction.

finishes; otherwise, stale key versions may reappear in the key ranges still undergoing recovery. Fortunately, because compaction is a background process, we can wait for recovery without affecting client operations.

Step (1) is implemented entirely within RocksDB. A RocksDB compaction iterator will record corrupted key ranges as they are encountered and withhold them from appearing in the compaction's result. At the end of the iterator's lifetime, ZippyDB is notified about the corrupted key range. Note that the compaction may encounter multiple corrupt key ranges, which are batched into a single patch request.

In step (2), the patch is reported to the primary. Step (3) must go through the primary because the primary is the only shard that can propose entries to the Paxos log. Note this does not mean primaries cannot recover from corrupted data blocks. The patch request in the Paxos log is simply a no-op that reserves a point of reference for the recovery procedure and includes information necessary for recovery, such as the corrupted key ranges and the ID of the corrupted shard. Any replica that encounters the patch request in the log is by definition up-to-date to that point in the Paxos log, which means any replica that isn't the corrupted replica is eligible to send a patch to the corrupted replica. In step (4), ZippyDB waits for the Paxos log to replicate the Paxos entry as well as for other replicas to consume the log until they encounter the patch request.

In step (5), an uncorrupted replica assembles a patch on the specified key range(s) with a RocksDB iterator. To do this, the uncorrupted replica opens a range scan iterator on each key range. Note that this replica might encounter a bit corruption while assembling the patch. In practice the probability of this is small because the number of keys covered by the patch is on the order of kilobytes (§5.2), and any scans to find such keys would predominantly look through metadata blocks, such as bloom filter or index blocks. However, if a replica does encounter a corruption while assembling a patch, it simply does not send a patch. Therefore, for the patch request to fail, *both* (or more, if the replication factor is more than 3) uncorrupted replicas will have to encounter a bit corruption, and this probability is low.

Step (6) is also implemented at the RocksDB level. When a replica applies a patch, simply inserting all the key-value pairs present in the patch is insufficient because of deleted keys. In particular, any key present in the requested key range and *not* present in the patch is an implicit delete. Therefore, to apply a patch, the corrupted shard must also delete any keys that it can see that aren't present in the patch. This case is possible because RocksDB deletes keys by inserting a tombstone value inline in SST files, but such a tombstone might have already been compacted away on the replica providing the patch. Hence the corrupted data block may contain tombstone operators that delete a key, and these must be preserved.

4.2.5 Transactions and Invalidating Snapshots

ZippyDB uses RocksDB snapshots to execute transactions. RocksDB snapshots are represented by a sequence number, s. Then, for as long as the snapshot s is active, RocksDB will not compact any version, s', of a key where s' is the greatest version of the key such that $s' < s$. If RocksDB invalidates a snapshot, then the ZippyDB transaction using that snapshot will abort and retry.

A subtle side-effect of a corrupted data block is snapshot corruption. For example, suppose the RocksDB store has a snapshot at sequence number 100 and the corrupted data block contains a key with sequence number 90. Because the data block is corrupted, it cannot be read, so we do not know whether the corruption affects snapshot 100. Then for safety, we need to invalidate all local snapshots, because any of them could have been affected by the corrupted key range. In practice, this is reasonable because most ZippyDB transactions (and hence RocksDB snapshots) have short lifetimes.

More generally, any transactional system that relies on versioning (e.g., MyRocks that is built on RocksDB), through either optimistic concurrency control or multi-version concurrency control (MVCC) can similarly deal with corruptions by aborting ongoing transactions.

4.3 Cascading Errors

In both HDFS-DIRECT and ZippyDB-DIRECT, the system will visit multiple replicas if necessary to resolve a bit error. However, currently DIRECT ignores and does not try to fix any cascading errors encountered during this process. For example, if a replica tries to assemble a patch in ZippyDB-DIRECT and fails because the iterator encounters a corruption, the replica will simply cleanup, ignore the patch request, and move to executing the next request in the Paxos log. We ignore cascading errors for simplicity but can easily incorporate recovery of cascaded errors in the future.

5 Evaluation

This section evaluates DIRECT by answering the following questions: (1) By how much does DIRECT decrease application-level errors in both HDFS and ZippyDB? In HDFS, how far can DIRECT drive UBER while avoiding application-level errors? (2) How much does DIRECT decrease time to recovery from corruption errors in ZippyDB?

Note we do not measure recovery time in HDFS because DIRECT handles bit errors *synchronously*, which means read errors only propagate to the application level if DIRECT cannot fix them. Therefore, "recovery time" can be measured by its effect on read latency. On the other hand, in ZippyDB, DIRECT handles bit errors asynchronously because recovery procedures must go through the coordination layer, as described in Section 4.2.

Experimental Setup. To evaluate ZippyDB, we set up a cluster of 60 Facebook servers that capture and duplicate live traffic from a heavily loaded service used in computing

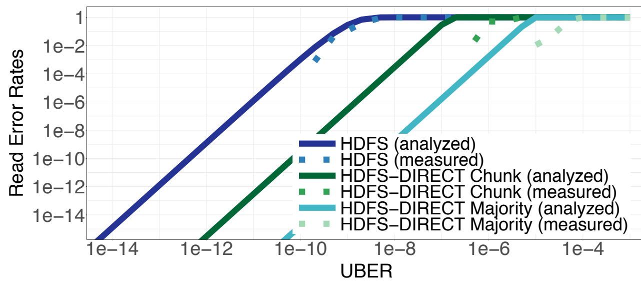


Figure 7: Read error rate for HDFS with varying UBER. HDFS-DIRECT Chunk is based on chunk-by-chunk recovery, while HDFS-DIRECT Majority is computed on bit-by-bit majority. The analyzed data is computed using the formulas in §3.

user feeds. To evaluate HDFS, we set up an HDFS cluster on machines with 8 ARMv8 cores at 2.4 GHz, 64 GB of RAM, and 120 GB of flash. The cluster has three DataNode machines, and four machines act as HDFS clients. The machines are connected with 10Gb links. HDFS experiments have a load and read phase: in the load phase, we load the cluster with 500, 128MB files with random data. In the read phase, clients randomly select files to read. After the load phase, we clear the page cache.

Error Injection. To simulate UBERs, we inject bit errors into the files of both systems. In ZippyDB, we inject errors with a custom RocksDB environment that flips bits as they are read from a file. In HDFS, we run a script in between the load and read phases that flips bits in on-disk files and flushes them. For an UBER of μ, e.g. $\mu = 10^{-11}$, we inject errors at the rate of 1 bit flip per $1/\mu$ bits read. We tested with UBERs higher than the manufacturer advertised 10^{-15} to test the system's performance under high error rates, and so that we can measure enough bit errors during an experiment time of 12 hours rather than several days (or years)[3].

Baselines. We compare against unmodified HDFS and ZippyDB, both systems used in production for many years. Although unmodified HDFS does compute checksums for chunks, it does not recover at that granularity. HDFS-DIRECT leverages these checksums during recovery, which allows it to recover blocks synchronously within client reads. In unmodified ZippyDB, when a RocksDB instance encounters a compaction error, the entire ZippyDB server crashes. While this may seem like an overly aggressive baseline, due to the difficulty of recovering an individual object in a logically-replicated system, we did not find an alternative baseline that was correct and easier to implement. One possible strawman is to recover the individual RocksDB instance that encountered a bit error. Even this approach has significant error amplification (tens of GBs per bit error), and suffers from high implementation complexity, as ZippyDB has no existing logic for recovering individual RocksDB instances.

[3] Note that an UBER 10^{-11} is $10,000\times$ *higher* than 10^{-15}.

UBER	HDFS Thruput [GB/s]	HDFS-DIRECT Thruput [GB/s]
10^{-7}	0.00 ± 0.00	2.09 ± 0.08
10^{-8}	0.00 ± 0.00	2.56 ± 0.09
10^{-9}	2.46 ± 0.08	2.55 ± 0.07
10^{-10}	2.89 ± 0.10	2.84 ± 0.07
No errors	2.83 ± 0.07	2.88 ± 0.07

Table 2: Throughput of HDFS and HDFS-DIRECT. At UBER of 10^{-8}, HDFS throughput collapses due to bit errors.

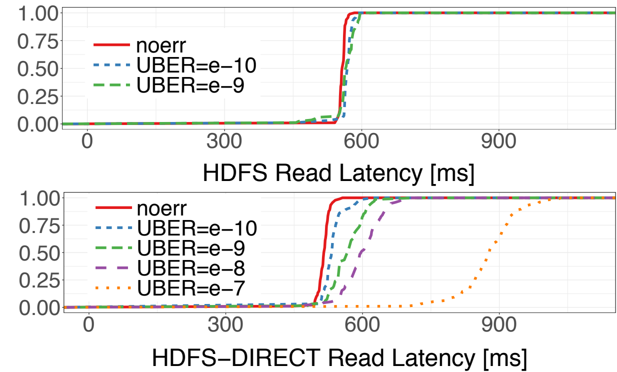


Figure 8: Read latencies (128 MB) of HDFS and HDFS-DIRECT. All reads fail in HDFS an UBER of 10^{-8} and higher. (Note that all latencies in HDFS-DIRECT are shifted slightly to the left, and this is due to temporal variations in our shared, experimental testbed.)

5.1 HDFS

UBER Tolerance. The main advantage of HDFS-DIRECT over HDFS is the ability to tolerate much higher UBERs with chunk-level recovery and majority voting. Figure 7 reports block read error rates of HDFS with varying UBERs. In HDFS, read errors are also considered *data loss*, because the data is unreadable (and hence unrecoverable) even after trying all 3 replicas. The figure shows the measured read error on our HDFS experimental setup, within the UBER range in which we could effectively measure errors, as well as the computed read error based on the computation presented in §3. We compared unmodified HDFS, with chunk-by-chunk recovery and bit-by-bit majority. The experimental read error is collected by running thousands of file reads and measuring how many fail. The measured results are relatively close to the analytical results, and in fact experience even fewer errors than the analytical model (the measured curves are shifted to the right of the analytic curves). We believe the primary reason is that the Taylor's approximation used in the analytical model does not hold for high UBERs. As expected, bit-by-bit majority (green lines) reduces the read error rate even further due to its lower error amplification (it can recover bit-by-bit). Both our analysis and the experimental results show that HDFS-DIRECT can tolerate a $10,000\times$–$100,000\times$ higher UBER and maintain the same read error rate.

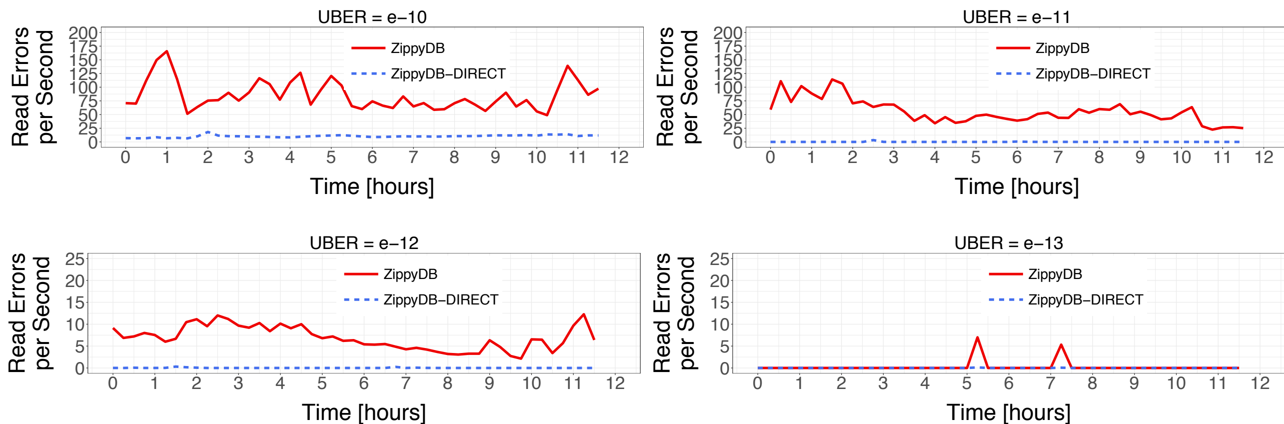


Figure 9: Read error rates over time in ZippyDB and ZippyDB-DIRECT, under varying UBERs.

UBER	Time to Complete Benchmark (s)
10^{-7}	177.4 ± 2.5
10^{-8}	169.4 ± 2.1
No errors	166.2 ± 1.8

Table 3: Time for HDFS-DIRECT to complete TeraSort benchmark. Note that we do not present results for unmodified HDFS, because for the UBERs tested, HDFS cannot complete any reads.

Overhead of DIRECT. Because DIRECT corrects bit errors synchronously in HDFS, error correction poses an overhead on reads that encounter bit errors. Table 2 shows the throughput of both systems, measured by saturating the DataNodes with four, 64-threaded clients that are continuously reading random files. The throughput of HDFS goes to zero at an UBER of 10^{-8}, because it cannot complete any reads due to corruption errors. Such failures do not occur in HDFS-DIRECT, although its throughput decreases modestly due to the overhead of synchronously repairing corrupt chunks during reads.

For HDFS-DIRECT, we are also interested in latency incurred by synchronous chunk recovery. We compare the CDF of read latencies of 128 MB blocks for different UBERs in Figure 8. The higher the UBER, the more chunk recovery requests that need to be made during a block read and the longer these requests will take. The results in Figure 8 (and Table 2) highlight the fine-grained tradeoff between performance and recoverability that is exposed by DIRECT. We also report HDFS read latencies, but there is little difference across UBERs because only latency for successful block reads are included; again, we do not report results for UBERs higher than 10^{-8}, since at those error rates HDFS cannot successfully read any blocks.

We also note that the latencies reported are the time it takes to read an entire 128 MB *file*, which is composed of many (64K) chunks. Hence Figure 8 hides a small *chunk* tail latency introduced by DIRECT. For example, chunks can now encounter errors on 0, 1, 2, or 3 of its replicas. Most chunks will encounter 0 errors, but some may encounter errors on all 3 of its replicas, which will require a relatively costly majority voting round. However, these disparate chunk latencies are hidden in the file latency, because all files have almost the expected number of errors (128 MB$\cdot UBER$).

Interestingly, these overheads become minimal when we run an end-to-end benchmark. We ran the TeraSort benchmark, a canonical Hadoop benchmark. We configured TeraSort to generate and sort 20 GB of data. Table 3 shows the time it takes HDFS-DIRECT to complete the TeraSort benchmark. Note that at an UBER of 10^{-8}, the time it takes to complete the benchmark is similar to when there are no errors (in fact, we do not report results for UBERs lower than 10^{-8} because they are so similar to results when there are no errors). Even at an UBER of 10^{-7}, the performance overhead is relatively low, because TeraSort is dominated by sort time in the mappers and reducers, rather than the time it takes to read the data into memory. These results suggest that even at very high UBERs, DIRECT imposes a low recovery overhead in workloads that are not disk-bound.

5.2 ZippyDB

UBER Tolerance. One main difference between unmodified ZippyDB and ZippyDB-DIRECT is that ZippyDB-DIRECT avoids crashing when encountering a bit error. To characterize how many server crashes are mitigated with DIRECT, we measured the average rate of compaction errors per hour *per server*, over 12 hours. The results are shown in Table 4. Figure 9 shows the read error rate over time of both systems, and Table 4 also shows the number of read errors as a percentage of all reads. Note that the error rate patterns across UBERs are different because they are run during different time intervals, so each UBER experiment sees different traffic. We did try to ensure read/write QPS and query distribution remain steady throughout the experiments[4].

The error rate is much higher for ZippyDB than ZippyDB-DIRECT because not only do clients see errors from regular read operations, but also they experience the spike in errors when a server shuts down due to a compaction corruption.

[4]Unfortunately, there is no tracing system set up for ZippyDB, so we were unable to capture and replay traces for consistency.

	Read Errors		Compaction Errors per Hour per Server
UBER	ZippyDB	ZippyDB-DIRECT	ZippyDB
10^{-10}	2.7308%	0.1865%	0.1991 ± 0.1077
10^{-11}	1.9808%	0.0400%	0.0621 ± 0.0455
10^{-12}	0.2650%	0.0008%	0.0038 ± 0.0035
10^{-13}	0.0108%	0.0002%	0.0003 ± 0.0005

Table 4: Read and compaction errors with ZippyDB and ZippyDB-DIRECT. The read errors are a percentage of the total number of reads, and the compaction errors are the number of errors per hour per server. ZippyDB-DIRECT is able to fix all compaction errors in our experiment, while the server crashes in ZippyDB.

Time Spent in Reduced Durability. With DIRECT, we also seek to minimize the amount of time spent in reduced durability to decrease the likelihood of simultaneous replica failures. Figure 10 shows a CDF of the time it takes to recover from compaction errors in ZippyDB-DIRECT. The graph shows the amount of time it takes for replicas to process the Paxos log up until the patch request, as well as the overhead of constructing and inserting the patch. With DIRECT, this recovery time is on the order of *milliseconds*. In contrast, the period of reduced durability in unmodified ZippyDB due to a compaction error is on the order of *minutes*, depending on the amount of data stored in the crashed ZippyDB server. This is due to the high error amplification of ZippyDB, which invalidates 100s of RocksDB shards due to a single compaction bit error. With DIRECT, ZippyDB can reduce its recovery time due to a bit error by around $10,000\times$. Even with a more reasonable baseline that only invalidates the single RocksDB shard that experienced the error, we estimate that DIRECT can reduce the recovery time by around 100x.

We also found that the recovery latency is dependent on the size of the patch required to correct the corrupted key range. Figure 11 presents a CDF of the size of the patches generated during the recovery process. Patch size is also interesting because the recovery mechanism described in Section 4.2.4 recovers a *range* of keys, since the exact keys on the corrupted data block are impossible to identify. As we see in Figure 11, even though recovering a range can in theory increase error amplification, the number of keys required for recovery is still low (a single RocksDB instance contains on the order of millions of key-value pairs). Figure 11 also confirms that, generally, as the UBER increases, patch sizes increase due to more key ranges getting corrupted during a single compaction operation. We note that UBER=10^{-12} yielded an anomalous line, but by the time we analyzed the data, we no longer had access to the experimental system to rerun the results. We speculate that a variety of factors could have caused the anomaly: 1) The corruption error could have occurred on a particularly dense subset of the key space. 2) the corruption error might have occurred during a read to a bottom-level SST file in RocksDB; due to compaction, key-ranges used for recovery grow progressively larger in lower levels of the RocksDB LSM tree.

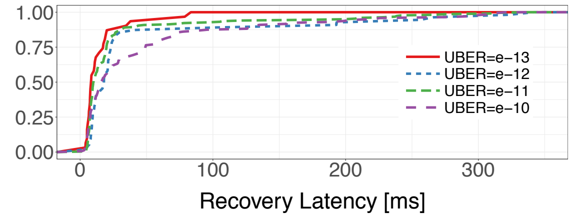


Figure 10: CDF of compaction recovery latencies in ZippyDB-DIRECT. ZippyDB-DIRECT takes milliseconds to recover from corruptions, while ZippyDB takes *minutes*.

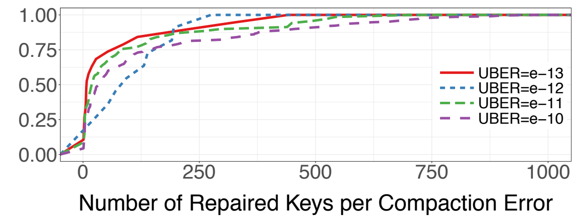


Figure 11: CDF of patch sizes generated during the ZippyDB-DIRECT recovery process. The patch size is small, which means low error amplification.

Measuring Cost. DIRECT trades off higher reliability and longer device lifetime for engineering and operational costs. We are unable to capture these costs in our evaluation but describe the changes to the software stack needed in order to run DIRECT in production. For example, ZippyDB-DIRECT required around 1200 lines of C++ code and HDFS-DIRECT required around 450 lines of Java code. As we discuss in Section 6, a DIRECT stack will also require running a hardened filesystem, such as ZFS, so that the local filesystem can continue functioning after encountering bit errors.

6 Discussion

Local File System Error Tolerance. When devices exhibit higher UBERs, local file systems also experience higher UBERs. DIRECT protects application-level metadata and data, which are data blocks to the local file system. Protecting local file system metadata such as inodes, the FS journal, etc. is beyond the scope of this paper. Several existing file systems protect metadata against bit corruptions [3, 17, 18, 43, 55, 70, 81]. The general approach is to add checksums and locally replicate for error correction. Another approach is to use more reliable hardware for metadata and less reliable hardware for data blocks [55].

Support for DIRECT. Some simple device-level mechanisms would help datacenter operators run devices past their manufacturer defined UBER. First, it would be beneficial if devices have a less aggressive "bad block policy", which is a firmware protocol for retiring blocks once they reach some heuristic-defined level of errors.

Second, it would be helpful if devices return the content of corrupted pages, although this is not a hard requirement This enables distributed storage applications to minimize recovery amplification by recovering data at a granularity smaller than

a device page. For example, to use majority voting, a system operator must use devices that return the content of corrupted pages, such as Open-Channel SSDs [25]. Fortunately, majority voting is optional and only applicable to block-replicated systems, and all other aspects of DIRECT apply if the system operator uses traditional flash devices. In case corrupt pages cannot be read, copies of local metadata must be stored on separate physical pages. Otherwise, a page error could invalidate all copies of the metadata.

Retrofitting PostgreSQL. As we discussed earlier, PostgreSQL is difficult to retrofit with DIRECT. This is because Postgres pages do not have indexing information; indexes in Postgres are stored on separate pages, if at all (no indexes are built without explicit user commands). Postgres checksums are at the page granularity, so if there is a bit error on a data page, DIRECT would need to figure out all the tuples stored on the page in order to both minimize error amplification and do correct recovery (Postgres uses MVCC to support transactions). The only way to determine these tuples and their versions is to comb through the index pages for any pointers to the corrupt page: in particular, we observe that what we need for DIRECT is a *reverse* index, one that maps from pages to tuples, rather than from tuples to pages. Generally, for DIRECT to be efficient in logically-replicated systems, the page layout must provide insight into what tuples are stored on a page. For example, RocksDB builds such a reverse index implicitly in the index blocks of its file format.

Network Partitions. Because DIRECT uses remote redundancy to correct bit errors, network failures can now affect the recovery process. Fortunately, real-world studies have shown that the most common kind of network failure— link failures — do not greatly affect application availability, because there are enough redundant paths built into the network topology [41]. In future work, we plan to both model and evaluate how transient or permanent network failures affect both recovery latency and error rate.

7 Related Work

Our work departs from existing work on data integrity in data storage systems [24, 26, 53, 73] because we expose bit corruptions at the distributed layer, rather than containing them in the storage layer. Furthermore, DIRECT does not stop at identifying corruptions but introduces a principled and performant way of fixing them to achieve high availability.

Software-level Redundancy. DIRECT is related to PAR [20] and PASC [33], which demonstrate how consensus-based protocols can be adapted to address bit-level errors. Unlike both of these works, which only address consensus protocols, our work tackles bit-level errors in general purpose storage systems. We also show how increasing the resiliency to bit errors can significantly reduce storage costs and improve live recovery speed in datacenter environments.

Other related work use different approaches. HARDFS [35] hardens local HDFS nodes by augmenting each node with a lightweight version that verifies its behavior. HDFS-DIRECT generalizes HARDFS, by only applying local protection to metadata and leveraging distributed replicas to recover data. FlexECC [46] and Duracache [60] are flash-based key-value caches that use less reliable disks by treating devices errors as cache misses. D-GRAID is a RAID storage system that gracefully degrades by minimizing the amount of data needed to recover from bit corruptions [74]. AHEAD and EDB-Tree apply software-level error detection and correction to address DRAM corruption in databases [51, 52].

There is a large number of distributed storage systems that use inexpensive, unreliable hardware, while providing consistency and reliability guarantees [23, 34, 40]. However, these systems treat bit corruptions similar to entire-node failures and suffer from high recovery amplification.

Hardware-level Redundancy. Several studies explore extending SSD lifetime via more aggressive or adaptive hardware error correction. Tanakamuru *et al.* [77] propose adapting codeword size based on the device's wear level to improve SSD lifetime. Cai *et al.* [30] and Liu *et al.* [61] introduce techniques to dynamically learn and adjust the cell voltage levels based on retention age. Zhao *et al.* [83] propose using the soft information with LDPC error correction to increase lifetime. Our approach is different: instead of improving hardware-based error correction, we leverage existing software-based redundancy to address bit-level errors.

8 Conclusion and Future Work

This paper presents DIRECT, a set of general techniques that use the inherent redundancy that exists in distributed storage applications for live recovery of bit corruptions. We showed with implementations of DIRECT in HDFS and ZippyDB that these techniques are widely applicable and, once implemented, can increase the bit error rate tolerance of distributed systems by orders of magnitude.

We envision extending the DIRECT approach in several directions. First, distributed storage systems can control error correction depending on how sensitive particular data is to bit corruptions (e.g. critical metadata). Second, distributed storage systems can control hardware mechanisms that influence the reliability as well as the performance of the device. For example, storing fewer bits per cell may reduce the latency of the device (at the expense of its capacity), and offer higher reliability. Certain applications may prefer to use a hybrid of low latency and low capacity devices for hot data, while reserving the high capacity devices for colder data.

9 Acknowledgements

We thank Mikhail Antonov, Siying Dong, Kim Hazelwood, Binu John, Chris Petersen, Muhammad Waliji, and the extended ZippyDB and RocksDB teams for their support on this project. We also thank our shepherd, Bianca Schroeder, and the anonymous reviewers for their excellent feedback.

References

[1] CockroachDB docs. https://www.cockroachlabs.com/docs/stable/.

[2] DRAM prices continue to climb. https://epsnews.com/2017/08/18/dram-prices-continue-climb/.

[3] Ext4 metadata checksums. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/.

[4] High-efficiency SSD for reliable data storage systems. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T2A_Yang.pdf.

[5] Introducing Lightning: A flexible NVMe JBOF. https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/.

[6] LevelDB. http://leveldb.org.

[7] McDipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[8] Micron 5210 ION SSD. https://www.micron.com/solutions/technical-briefs/micron-5210-ion-ssd.

[9] MongoDB docs. https://docs.mongodb.com/.

[10] Novel 4k error correcting code for QLC NAND. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FE22_Kuo.pdf.

[11] Project Voldemort: A distributed key-value storage system. http://www.project-voldemort.com/voldemort.

[12] RocksDB. http://rocksdb.org.

[13] RocksDB block based table format. https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format.

[14] SanDisk datasheet. https://www.sandisk.com/business/datacenter/resources/data-sheets/fusion-iomemory-sx350_datasheet.

[15] Under the hood: Building and open-sourcing RocksDB. http://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920.

[16] What is R.A.I.S.E? https://www.kingston.com/us/ssd/raise.

[17] XFS reliable detection and repair of metadata corruption. http://xfs.org/index.php/Reliable_Detection_and_Repair_of_Metadata_Corruption.

[18] The Z File System (ZFS). https://www.freebsd.org/doc/handbook/zfs.html.

[19] NAND flash media management through RAIN. Technical report, Micron, 2013.

[20] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32, Oakland, CA, 2018. USENIX Association.

[21] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX Annual Technical Conference*, pages 91–102, 2013.

[22] P. Alcorn. Facebook asks for QLC NAND, Toshiba answers with 100TB QLC SSDs with TSV. http://www.tomshardware.com/news/qlc-nand-ssd-toshiba-facebook,32451.html.

[23] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[24] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[25] M. Bjørling, J. González, and P. Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[26] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Dealing proactively with data corruption: Challenges and opportunities. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 34–39. IEEE, 2011.

[27] D. Borthakur. HDFS block replica placement in your hands now! http://hadoopblog.blogspot.com/2009/09/hdfs-block-replica-placement-in-your.html.

[28] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for data centers. Technical report, Google, 2016.

[29] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526, Dresden, Germany, 2012.

[30] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, pages 551–563, San Francisco, CA, 2015.

[31] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[32] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, 2013.

[33] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Berkeley, CA, USA, 2012. USENIX Association.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220, Oct. 2007.

[35] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 105–118, San Jose, CA, 2013. USENIX.

[36] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.

[37] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13, 2018.

[38] D. Exchange. DRAM supply to remain tight with its annual bit growth for 2018 forecast at just 19.6%. `www.dramexchange.com`.

[39] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[40] S. Ghemawat, H. Gobioff, and S.-T. 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. ACM.

[41] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, 2011.

[42] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 17–24, San Jose, CA, 2012.

[43] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 293–306, New York, NY, USA, 2007. ACM.

[44] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 199–212, Santa Clara, CA, 2014.

[45] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure Storage. In *Usenix annual technical conference*, pages 15–26. Boston, MA, 2012.

[46] P. Huang, P. Subedi, X. He, S. He, and K. Zhou. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 489–500, Philadelphia, PA, 2014.

[47] J. Jeong, S. S. Hahn, S. Lee, and J. Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *FAST*, pages 61–74, 2014.

[48] K. Kambatla and Y. Chen. The truth about MapReduce performance on SSDs. In *Proceedings of the 28th Large Installation System Administration Conference*, pages 118–126, Seattle, WA, 2014.

[49] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The memory forum*, pages 1–4, 2014.

[50] H. Kim, S.-J. Ahn, Y. G. Shin, K. Lee, and E. Jung. Evolution of NAND flash memory: From 2D to 3D as a storage market leader. In *Memory Workshop (IMW), 2017 IEEE International*, pages 1–4. IEEE, 2017.

[51] T. Kolditz, D. Habich, W. Lehner, M. Werner, and S. T. de Bruijn. AHEAD: Adaptable data hardening for on-the-fly hardware error detection during database query processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1619–1634, New York, NY, USA, 2018. ACM.

[52] T. Kolditz, T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Online bit flip detection for in-memory B-trees on unreliable hardware. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 5:1–5:9, New York, NY, USA, 2014. ACM.

[53] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, volume 2008, page 127, 2008.

[54] J. Lee, J. Jang, J. Lim, Y. G. Shin, K. Lee, and E. Jung. A new ruler on the storage market: 3D-NAND flash for high-density memory and its technology evolutions and challenges on the future. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 11–2. IEEE, 2016.

[55] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[56] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 1–1. IEEE, 2016.

[57] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 501–512, 2014.

[58] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, Vancouver, BC, 2015.

[59] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011.

[60] R. Liu, C. Yang, C. Li, and G. Chen. DuraCache: a durable SSD cache using MLC NAND flash. In *Proceedings of the 50th Annual Design Automation Conference 2013*, pages 166–171, Austin,TX, 2013.

[61] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, San Jose, CA, 2012.

[62] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016.

[63] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 106–106, New York, NY, USA, 2018. ACM.

[64] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 177–190, Portland, Oregon, 2015.

[65] R. Micheloni et al. *3D Flash memories*. Springer, 2016.

[66] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[67] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale Internet storage systems. *SIGARCH Computing Architecture News*, 42(1):471–484, 2014.

[68] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[69] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, 1988.

[70] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[71] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proceedings of the 39th International Conference on Very Large Data Bases*, pages 325–336, Trento, Italy, 2013.

[72] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 67–80, Santa Clara, CA, 2016.

[73] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36. ACM, 2005.

[74] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *Trans. Storage*, 1(2):133–170, May 2005.

[75] A. S. Spinelli, C. M. Compagnoni, and A. L. Lacaita. Reliability of NAND flash memories: Planar cells and emerging issues in 3D devices. *Computers*, 6(2):16, 2017.

[76] B. Tallis. QLC NAND arrives for consumer SSDs. `https://www.anandtech.com/show/13078/the-intel-ssd-660p-ssd-review-qlc-nand-arrives`.

[77] S. Tanakamaru, M. Fukuda, K. Higuchi, A. Esumi, M. Ito, K. Li, and K. Takeuchi. Post-manufacturing, 17-times acceptable raw bit error rate enhancement, dynamic codeword transition ECC scheme for highly reliable solid-state drives, SSDs. *Solid-State Electronics*, 58(1):2–10, 2011.

[78] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 373–386, Santa Clara, CA, 2015.

[79] The Apache Software Foundation. Apache Cassandra. `http://cassandra.apache.org/`.

[80] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 328–337, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[81] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[82] G. Zemor and G. D. Cohen. Error-correcting WOM-codes. *IEEE Transactions on Information Theory*, 37(3):730–734, 1991.

[83] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 243–256, San Jose, CA, 2013.

Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems

Zhufan Wang[†], Guangyan Zhang[†,*] Yang Wang[‡], Qinglin Yang[†], Jiaji Zhu[§]
[†]*Tsinghua University,* [‡]*The Ohio State University,* [§]*Alibaba Cloud*

Abstract

This paper tries to accelerate data recovery in a large-scale storage system with minimal interference to foreground traffic. By investigating I/O and failure traces from a real-world large-scale storage system, we find that because of the scale of the system and the imbalanced and dynamic foreground traffic, no existing recovery protocols can generate a high-quality recovery strategy in a short time.

To address this problem, this paper proposes Dayu, a timeslot-based recovery protocol, which only schedules a subset of tasks which are expected to finish in one timeslot: this approach reduces the computation overhead and naturally can cope with the dynamic foreground traffic. In each timeslot, Dayu incorporates four key algorithms, which enhance existing solutions with heuristics motivated by our trace analysis.

Our evaluations in a 1,000-node real cluster and in a 25,000-node simulation both confirm that Dayu can outperform existing recovery protocols, achieving high speed and high quality.

1 Introduction

This paper describes our experience and methods to accelerate data recovery in Pangu [1] , a real-world large-scale storage system with 10K nodes and tens of TBs of storage per node.

As a cloud storage provider, AliCloud, the owner of Pangu, needs to make a promise of data durability to its customers (*i.e.*, the chance of data loss is smaller than a threshold). For marketing reasons, the owner has a strong motivation to improve data durability, so that its promise can be appealing compared to its competitors. This motivates us to investigate whether it is possible to accelerate data recovery in Pangu, because recovery speed is one of the determining factors of data durability [2].

Similar to previous works [3–7], Pangu divides data into chunks (usually tens of MBs), replicates these data chunks, and distributes these replicas to different nodes. When a node fails, Pangu re-replicates its data chunks: since the replicas of these chunks are distributed to different nodes, Pangu asks all these nodes to copy chunks in parallel [4, 8].

To re-replicate data chunks of the failed node, the recovery protocol needs to schedule a source, a destination, and a bandwidth for each of these data chunks. An ideal scheduling algorithm should achieve at least the following two goals: first, the algorithm should generate a high-quality strategy, which should allow data re-replication to be completed as soon as possible under the constraint that it has minimal impact on foreground traffic; second, the speed of the scheduling algorithm itself should be high enough so that it does not become the bottleneck of data recovery.

To understand the quality and speed of existing scheduling algorithms, we analyze the failure and I/O traces from a real deployment of Pangu. We find none of the existing algorithms can achieve both acceptable quality and acceptable speed, because of the following challenges:

- *Very-large scale*: the largest deployment of Pangu has more than 10K nodes and up to 72 TBs of storage (about 1.5M chunks) per node. Therefore, when a node fails, the algorithm needs to decide how to recover all these data chunks and each chunk has about 10K nodes as candidate destinations.
- *Tight time constraint*: given the scale of the system, data chunks of a failed node can be re-replicated with a high degree of parallelism. Our simulation shows that if the idle bandwidth can be fully utilized, the recovery can be finished within tens of seconds, which means the scheduling algorithm itself should complete within seconds.
- *Imbalanced foreground traffic and available data*: we find a two-fold imbalance, which poses challenges to the quality of scheduling. First, a number of nodes can have significantly heavier foreground traffic than the others; and second, some nodes can have more data chunks available for re-replication.
- *Dynamic foreground traffic*: the foreground traffic can change dramatically over time. To cope with such dynamic traffic, the recovery protocol needs to adjust its

[*]Corresponding author: gyzh@tsinghua.edu.cn

plan when it observes a significant change in the foreground traffic, which again calls for fast scheduling.

Our simulation of existing scheduling algorithms shows that, on the one hand, simple and decentralized algorithms like random selection or best-of-two-random [9] can finish scheduling quickly (*i.e.*, high speed), but they often cause a small number of nodes to be overloaded, increasing the recovery time and impairing the performance of foreground traffic (*i.e.*, low quality). On the other hand, sophisticated and centralized algorithms, such as Mixed-Integer Linear Programming [10–12], can effectively utilize available bandwidth and avoid overloading a node (*i.e.*, high quality), but they can take prohibitively long to compute a plan given the scale of our target system (*i.e.*, low speed).

This paper proposes Dayu, a high-speed and high-quality recovery protocol for large-scale, imbalanced, and dynamic storage systems. The key idea of Dayu is motivated by the observation that, to cope with dynamic foreground traffic, we need to periodically monitor the foreground traffic and adjust the recovery plan: in such a design, scheduling for all data chunks of the failed node together is both computationally heavy and unnecessary, since the plan is likely to be adjusted later. Following this observation, Dayu incorporates a timeslot-based solution: it divides time into multiple slots, whose length is determined by how frequent the underlying storage system monitors and reports idle bandwidth; based on such report, Dayu tries to schedule a subset of chunks so that they can be re-replicated within the current timeslot; if the actual re-replication of some chunks takes longer than expected for whatever reason, Dayu will re-schedule them in the next timeslot.

This approach brings two benefits: first, it reduces the computation overhead of scheduling because in each timeslot, the algorithm only needs to schedule a subset of tasks (about one third on average in our experiments). Second, this solution can naturally cope with the dynamic foreground traffic because Dayu's decision is based on the information collected at the beginning of each timeslot.

To realize this idea, Dayu incorporates four key techniques, which enhance existing algorithms based on our observations:

- *Greedy algorithm with bucket convex-hull optimization to schedule tasks*: Dayu uses a greedy algorithm to iteratively choose the most under-utilized candidate as the source and destination for each task, till it finds enough tasks to fill a timeslot. To reduce the computation overhead, Dayu incorporates the convex-hull optimization [13] and further proposes a bucket approximation to reduce the size of the candidate set.
- *Prioritizing nodes with high idle bandwidth but few available chunks*: Our observation shows that such nodes are likely to get under-utilized, if the scheduling algorithm decides to replicate their chunks from other nodes. Therefore, Dayu enhances the aforementioned greedy algorithm with the following heuristic: if a chunk to be re-replicated has a replica in such a prioritized node, Dayu will assign the node as the source.
- *Iterative WSS to allocate bandwidth for each task*: To minimize the completion time of chosen tasks, Dayu enhances the weighted shuffle scheduling algorithm (WSS) [14]: in each iteration, Dayu uses WSS to identify the bottlenecks in the remaining tasks, assigns a weighted fair share of bandwidth to each task correspondingly, and removes the bottleneck tasks and allocated bandwidth.
- *Re-scheduling stragglers*: Straggler tasks will inevitably occur due to mis-prediction of the foreground traffic or unexpected hardware faults, so Dayu has to re-schedule them in the next timeslot. Straggler tasks are different from new tasks, since we prefer keeping their destinations unchanged: otherwise, we will lose their existing progress. Dayu first estimates whether it is worth changing their destinations, and then re-computes their sources and allocated bandwidth.

Our evaluation of Dayu on a real deployment of 1,000 nodes shows that, compared to Pangu, Dayu increases the recovery speed by 2.96× and increases the p90 latency (i.e., tail latency at 90th percentile) of the foreground traffic during recovery by only 3.7%. Our simulation shows that Dayu outperforms various existing solutions and can scale to a cluster of 25K nodes.

2 Background and Observations

2.1 Background of Pangu

Pangu is the underlying storage system of AliCloud, one of the largest public cloud providers in Asia [1]. Pangu inherits the classic distributed file system architecture from previous works like GFS [3], HDFS [5], Cosmos [6], and Azure [7]. It splits data into multiple chunks (the most common chunk size is 64MB) and stores data chunks on a large number of data servers called *ChunkServers*. A metadata server called *MetaServer* maintains the metadata of the distributed file system, such as the locations of data chunks. Given its very large scale, Pangu incorporates multiple *MetaServers*, each responsible for a subset of metadata [15–17]. Besides, Pangu incorporates a *RootServer* to route clients to the corresponding *MetaServer*. To achieve uniform data distribution, Pangu uses random or weighted random mechanism to place data on different *ChunkServers*.

Like most existing systems, Pangu replicates data chunks (most chunks have three replicas) so that if a node fails, Pangu can recover its data chunks by copying from other replicas. For each data chunk to be recovered, Pangu needs to choose a source and a destination for data copy: there are usually a few candidate sources depending on the number of replicas and a

large number of candidate destinations. The current version of Pangu randomly picks a source and a destination for each data chunk to be recovered.

In the current deployment of Pangu, we observe that the network bandwidth is usually the bottleneck when performing such data recovery: most Pangu nodes are equipped with 1Gb or 10Gb Ethernet, whose bandwidth is smaller than the aggregate disk bandwidth; the deployment of high-speed devices, such as Infiniband, is limited due to cost reasons. Pangu's core network switches are organized using CLOS topology or fat-tree topology [18–21], so that there is no oversubscription. The core-to-rack link may be oversubscribed depending on the configuration: if the link is oversubscribed, the rack switch is usually the bottleneck; if not, the NICs of end-hosts are the bottlenecks.

During data recovery, Pangu is still servicing foreground applications, which may contend for network bandwidth. To limit the interference of data recovery on foreground traffic, Pangu provides a mechanism to limit the bandwidth utilization of one or a group of links on a node. With this mechanism, we can set a limit on the bandwidth of the recovery traffic, depending on how much interference one is willing to tolerate and the bandwidth of the foreground traffic. In Dayu, we limit the bandwidth of the recovery traffic on each node to be

$$B_{recover} = max(\alpha \times B_{total} - B_{foreground}, B_{min}) \qquad (1)$$

In this equation, B_{total} is the total bandwidth of the node; $B_{foreground}$ is the bandwidth of the foreground traffic; and α is a parameter to control the interference of the recovery traffic on the foreground traffic. We set α to be 75% and our experiments show that using this setting will incur negligible impact on p90 latency of the foreground traffic. As a storage system mainly designed for large files, Pangu does not aim at optimizing extreme tail latency (e.g. 99.9 percentile [22]), so this setting can satisfy our requirement, and if one is targeting even smaller interference, he/she can further decrease α. B_{min} is the minimal bandwidth the node will assign for recovery, which is to ensure that recovery will not be too slow. Both Pangu and Dayu set B_{min} to 30MB/s.

2.2 Observations

In this subsection, we analyze the workload and data placement from one deployment of Pangu to understand how they affect data recovery. We acquire such information from a data center of approximately 3500 nodes, each with two 1G NICs and 11 2TB hard drives. In this case, the aggregate bandwidth of hard drives is larger than that of the NICs. The storage system mainly serves online data processing service (*ODPS*), including MapReduce and data query. What we analyze includes 1) a checkpoint of a *MetaServer* in April 2018, which records the metadata related to the size and distribution of the data chunks, and 2) the trace of the foreground traffic and background recovery traffic in the coming week. Unless otherwise noted, our simulation experiments are on this 3500-node cluster throughout this paper. We study the scalability of Dayu beyond 3500 nodes in Section 5.2.

We make the following observations from the analysis:

Observation 1 *Each node stores hundreds of thousands of chunks.*

Figure 1(a) shows the CDF of the number of chunks on each node. We can observe that a majority of the nodes have around 250K chunks per node. This observation suggests two things: first, a recovery protocol needs to schedule how to recover so many chunks when a node fails. Second, when one node fails, each of the remaining nodes will participate in the re-replication of about 70 chunks on average (250K/3500).

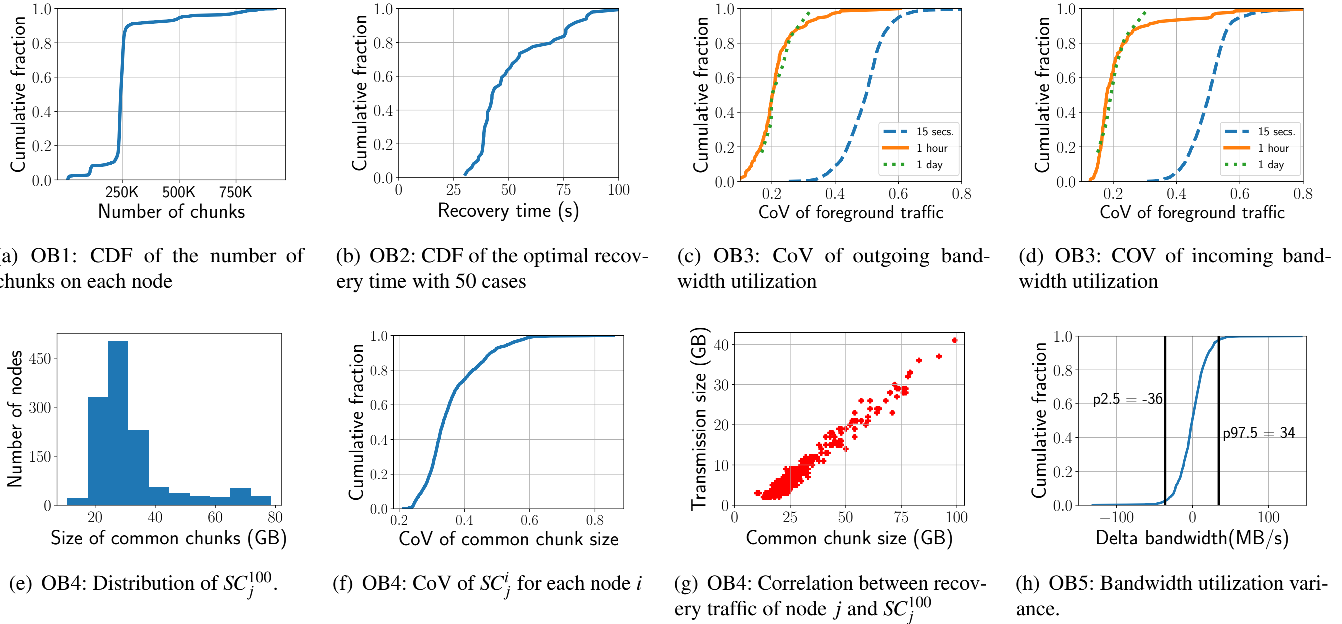
Observation 2 *The foreground traffic consumes less than half of the bandwidth on average. If all available bandwidth (computed using Equation 1) can be used for recovery, the system can recover 250K chunks in 51 seconds on average.*

We calculate the optimal recovery time for 50 different cases, assuming all available bandwidth can be utilized, and present the CDF of the recovery time in Figure 1(b).

This observation suggests that, although there are a large number of chunks to recover for each node failure, the highly parallel recovery in a large-scale system can recover these chunks in a short time, which calls for fast scheduling during the recovery protocol. However, the actual recovery in the trace often takes 2-4 minutes, i.e. $2.35 - 4.70\times$ of the ideal recovery time, which motivates our further investigation.

Observation 3 *The foreground traffic is experiencing significant short-term load imbalance.*

The trace we analyze records the foreground bandwidth of each node every 15 seconds. To understand whether the foreground traffic is balanced, we compute the coefficient of variation (CoV, standard deviation as a percentage of the mean) of foreground bandwidth in each timeslot, which is a standard metric to measure the variation of values. Then we draw the distribution of CoVs of different timeslots in Figure 1(c) and Figure 1(d). As shown in this figure, the CoVs of most timeslots are between 0.4 and 0.6, which is quite significant. Interestingly, if we measure such imbalance in a coarser granularity (i.e. one hour and one day), the imbalance becomes much smaller. Such results indicate that the system is relatively load balanced in a long term, but more imbalanced in a short term, which creates a challenge for data recovery: traditional load balancing techniques, such as data migration, mainly targets long-term imbalance, because they cannot run very frequently; data recovery, however, is mainly affected by short-term imbalance, because it can finish within tens to a few hundred seconds. This observation suggests that our recovery protocol must take such short-term imbalance into consideration, without relying on load balancing techniques.



(a) OB1: CDF of the number of chunks on each node

(b) OB2: CDF of the optimal recovery time with 50 cases

(c) OB3: CoV of outgoing bandwidth utilization

(d) OB3: COV of incoming bandwidth utilization

(e) OB4: Distribution of SC_j^{100}.

(f) OB4: CoV of SC_j^i for each node i

(g) OB4: Correlation between recovery traffic of node j and SC_j^{100}

(h) OB5: Bandwidth utilization variance.

Figure 1: Observations from a 3500-node real-world system

Observation 4 *Replicas of chunks on a given nodes are distributed unevenly among the other nodes.*

To understand how replicas of chunks on a given node are distributed among the other nodes, we define SC_j^i as the size of the chunks held by both node i and node j, which shows how much data node j can provide as source during recovery if node i fails.

We first sample a specific node $i = 100$. Figure 1(e) shows the distribution of SC_j^{100} for different j values. We can see that the histogram of the distribution fits the bell curve: this is actually mathematically provable (i.e. Central Limit Theorems) if we assume chunk placement is random. To understand such imbalance in the whole cluster, for each node i, we calculate the CoV of all the SC_j^i values and then we draw the CDF of CoVs of all nodes in Figure 1(f). One can observe that for a large portion of nodes, the distribution of SC_j^i is not balanced.

To understand how such imbalance affects recovery, we simulate the failure of node 100 with Pangu's random node selection strategy (Figure 1(g)) and find that there is a strong correlation between the size of outgoing recovery traffic of node j and SC_j^{100}. That means a node with a few (many) common chunks with the failed node will do little (much) work during recovery, but if the node has much (little) available bandwidth, it will get under-utilized (overloaded).

Observation 5 *Foreground traffic usually fluctuates within 14.4% of max bandwidth, but sometimes can change dramatically.*

Figure 1(h) shows how one node's foreground traffic changes in 5 hours. The delta bandwidth is the difference of the average bandwidth utilization in two adjacent timeslots. We can observe that in more than 95% of the cases (between "p2.5" and "p97.5" in Figure 1(h)), the absolute delta bandwidth is lower than 36MB/s, which is 14.4% of the maximum bandwidth (250 MB/s since each node has two 1Gb NICs). However, in the remaining 5% cases, the delta bandwidth can reach up to two thirds of the maximum bandwidth. Although the percentage of such extreme cases is small, they frequently happen in recovery, because the highly parallel recovery usually involves many nodes. Our simulation shows that they can create stragglers in recovery and thus are one of the major reasons why recovery speed is not ideal.

3 Dayu Overview

We call the re-replication of one data chunk a "recovery task" in the rest of the paper. Dayu achieves fast data recovery and low application interference by introducing a centralized scheduler called an *ObServer*, which performs timeslot-based recovery task scheduling.

Dayu assumes all the data servers periodically report their chunk placement and network utilization to the *ObServer*, and all the metadata servers send information of the recovery tasks to the *ObServer*. The rest of this paper presents the *ObServer*'s scheduling algorithm, which decides the source, the destination, and the bandwidth of a recovery task.

To achieve high-speed and high-quality scheduling, the key idea of Dayu is to schedule recovery tasks in multiple batches, instead of scheduling all of them together. This design choice is motivated by several reasons: first, since each node is usually involved in tens of recovery tasks (Observa-

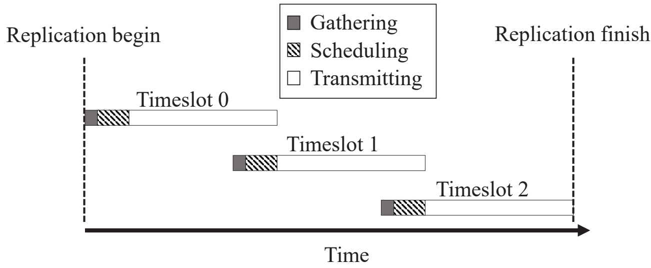


Figure 2: Timeslot-based scheduling in Dayu

$T_{timeslot}$	Length of a timeslot
$B^i_{recover_in/out}$	Node i's available incoming/outgoing bandwidth for recovery (by Equation 1)
s_t	Size of recovery task t
$c^i_{in/out}$	Total size of incoming/outgoing recovery tasks assigned to node i
α, β	Parameters

Table 1: Denotations

tion 1), scheduling tasks in multiple batches can still allow each node to fully participate in each batch and thus fully utilize its available bandwidth; second, scheduling tasks in batches can naturally cope with dynamic foreground traffic and infrequent measurement errors, because when observing any changes in the foreground traffic, Dayu can make adjustment in the next batch; finally, scheduling tasks in batches naturally reduces the computation overhead of the scheduling algorithm, because for each batch, the algorithm only needs to schedule a subset of tasks.

To implement this idea, as shown in Figure 2, Dayu divides the whole recovery time into multiple fixed-length time slices (called *timeslots* throughout the paper). At the beginning of a timeslot, the *ObServer* collects the latest state of the data servers. Using the state obtained, the *ObServer* chooses and schedules a subset of recovery tasks in this timeslot, including those recovery tasks scheduled in the last timeslot but unfinished yet. To fully utilize the available bandwidth, Dayu overlaps multiple timeslots so that the information gathering and task scheduling of slot n is executed before the end of slot $n-1$. The length of a timeslot is determined by how frequently the underlying storage system collects and reports state.

As mentioned in Section 2.1, the bottleneck of data recovery is either the NICs of the end hosts or the rack switch, and to simplify description, the following text assumes the NICs of the end hosts are bottlenecks, and one can easily extend it to support bottleneck rack switches. Table 1 lists the denotations used in the paper.

Goals. Dayu tries to achieve the following goals.

- **Goal 1: Utilize the available bandwidth as much as possible.** This is a natural goal to minimize the overall recovery time. If we were to fully utilize the available bandwidth in one timeslot, the total size (S) of the chunks that can be replicated in the timeslot would be:

$$S = min(\sum_{i \in Nodes} B^i_{recover_in}, \sum_{i \in Nodes} B^i_{recover_out}) \times T_{timeslot} \quad (2)$$

- **Goal 2: Finish as many tasks as possible in the target timeslot.** We hope that the scheduled tasks can actually finish within the target timeslot: otherwise, we have to re-schedule them again, which increases the computation overhead. This goal may look similar to the first one, but it is not: the first goal suggests us to oversubscribe the network bandwidth (i.e. schedule more tasks than the bandwidth can handle), so that if the foreground traffic drops, we can still utilize such extra available bandwidth; the second goal, however, suggests us to undersubscribe the network bandwidth so that if the foreground traffic increases, we can still finish the scheduled tasks. Therefore, Dayu has to make a trade-off between these two goals.
- **Goal 3: Minimize the chance of significant stragglers.** Because we cannot accurately predict the future foreground traffic, stragglers will inevitably occur. We prefer many small stragglers to a few significant stragglers, because many small stragglers can be re-scheduled and executed in parallel to minimize the recovery time. However, this goal obviously contradicts with the second goal, so Dayu has to make a trade-off as well.

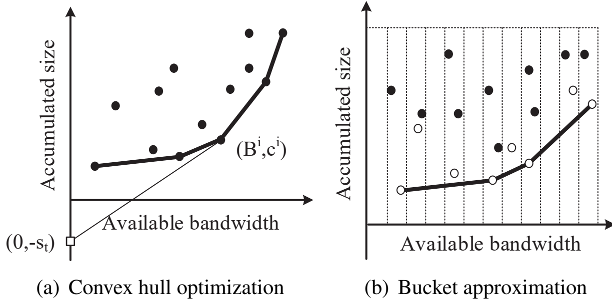
Overview of Dayu's algorithm. To achieve these goals, Dayu incorporates four key techniques, by enhancing existing algorithms with heuristics and approximations motivated by our observations: 1) a greedy algorithm with bucket convex hull optimization to select the source and the destination for each recovery task (§4.1); 2) a heuristic-based algorithm to prioritize nodes with a few common chunks with the failed node but a high available bandwidth (§4.2); 3) an iterative WSS algorithm to assign bandwidth for each task (§4.3); and 4) a heuristic-based algorithm to minimize the cost of re-scheduling straggler tasks (§4.4).

4 Design of Dayu

4.1 Selecting Source and Destination

Dayu iteratively scans all tasks and determines the source and the destination for each task, till it can find enough tasks to fill S (Equation 2). The candidate sources of a task include all nodes which hold a replica of the corresponding chunk; the candidate destinations of a task include all nodes which are not in the same rack of its sources.

To achieve the goals given in Section 3, Dayu incorporates a greedy algorithm: for each task, Dayu chooses the most under-utilized node in its candidate sources and destinations;



(a) Convex hull optimization (b) Bucket approximation

Figure 3: Reduce computation overhead with dynamic convex hull optimization. B^i and c^i are short for $B^i_{recover_in}$ and c^i_{in}.

if Dayu finds that even the most under-utilized candidate is going to be saturated, Dayu will skip this task.

The first question we need to answer is how to quantitatively measure the utilization of a node. We have tried several options, and through simulation, we decide to use the expected task finish time $\frac{c_{in/out}}{B_{recover_in/out}}$ ($c_{in/out}$ is the total size of the incoming/outgoing tasks assigned to this node) as the metric to evaluate the utilization of a node, because this metric achieves a nice balance between our first two goals.

Therefore, when choosing the source for task t, Dayu scans all its candidate nodes and chooses the one with the minimal $\frac{s_t+c_{out}}{B_{recover_out}}$ as the source. Such scanning is not computationally expensive, since most chunks have three replicas and one of them is already lost. Afterward, Dayu checks whether assigning task t to the source will saturate the source, i.e. its $\frac{s_t+c_{out}}{B_{recover_out}} > T_{timeslot}$: if so, Dayu will drop task t, because this means there is no way to complete task t in this timeslot.

Likewise, when choosing the destination for task t, Dayu chooses the node with the minimal $\frac{s_t+c_{in}}{B_{recover_in}}$. However, naively scanning all candidate destinations is computationally heavy, since the number of candidate destinations is large. To make things worse, greedy algorithms cannot be parallelized because each iteration depends on the result of the previous iteration. Our simulation on a 3500-node cluster shows that naively scanning all candidate destinations for each task can only achieve a speed of less than 30,000 tasks per second. Since our statistics shows that in one timeslot, Dayu can usually complete 60,000—150,000 tasks, this means naively scanning itself will take 2-5 seconds, which is not ideal. To address this challenge, we incorporate the dynamic convex hull optimization to accelerate this computation.

One can refer to [13] for the formal description of the convex hull optimization, and here we present an intuitive description. For each surviving node i, we draw a point $(B^i_{recover_in}, c^i_{in})$ in Cartesian coordinate system, as shown in Figure 3(a). Then for task t with size s_t, we draw another point $(0, -s_t)$ in Figure 3(a). Afterward, we draw a line from $(0, -s_t)$ to each other point: since the slope of each line is $\frac{c^i_{in}+s_t}{B^i_{recover_in}}$, finding the destination node for task t is equivalent to finding the line with the lowest slope.

We can maintain a dynamic convex hull to quickly search the line with the lowest slope. In a two-dimensional space, a convex hull is like a rubber band that wraps all the points tightly, where the lower convex shell is the lower part of this convex hull. We refer the point set of the lower convex shell as H (here we connect points in H together to form the lower convex shell as shown in Figure 3(a)). The points in H are connected counterclockwise. Then for a point p_h with precursor point and successor point in set H, the slope of line $p_{h-1} \rightarrow p_h$ must be less than or equal to the slope of line $p_h \rightarrow p_{h+1}$. We can find the node c in set H whose connection with point $(0, -s_t)$ has the smallest slope using binary search and the time complexity is $O(\log|H|)$.

After Dayu assigns task t to node i, its c^i_{in} is incremented by s_t. Therefore, Dayu needs to adjust the point of node i as well as the lower convex shell H: when a point p_h in H moves up, Dayu identifies the precursor (p_{h-1}) and successor (p_{h+1}) of p_h in original H, and scans all the points between them to find the new member(s) of H. The convex hull optimization reduces the complexity of scanning destination nodes from linear to sub-linear, without affecting the results of the greedy algorithm.

We further propose an approximate solution to reduce the candidate set of the lower convex shell, in turn boosting the speed of the algorithm. As shown in Figure 3(b), we divide the range of available incoming bandwidth into multiple equal-sized buckets. If nodes i and j are in the same bucket, they are considered to have approximately identical available bandwidth, i.e. $B^i_{recover_in} \approx B^j_{recover_in}$. Without loss of generality, we suppose $c_i > c_j$. Then node i cannot be the member of the lower convex shell. Therefore, only the lowest node within the same bucket can become the member of the lower convex shell. All those lowest nodes (hollow circles in Figure 3(b)) form a reduced candidate set, denoted C. We can construct the convex shell H from this reduced candidate set C, instead of the full set of nodes. After Dayu assigns a task to a node, it adjusts the point of this node as well as the reduced candidate set.

The bucket size determines the reduction degree of the bucket approximation. We use 1 MB/s as the bucket size in our experiments, and our simulation shows an average reduction factor of 22.8, and as a result, Dayu can complete selecting sources and destinations for about 210,000 chunks within one second—this is seven times faster than naive scanning.

Such bucket approximation certainly brings inaccuracy to the greedy algorithm, but such inaccuracy already exists as a result of measurement errors and fluctuation of foreground traffic. Therefore, as long as the bucket size is small, our approximation should not significantly increase such inaccuracy.

4.2 Prioritizing Underemployed Nodes

Our simulation on our greedy algorithm reveals the same problem as our Observations 3 and 4: nodes with high available bandwidth but only a small number of available chunks are likely to get under-utilized, which violates our first goal. We call them *underemployed nodes* in the rest of the paper. For example, suppose node A has an available outgoing bandwidth of 50MB/s and can be the source of Tasks 1 and 2; node B has an available outgoing bandwidth of 60MB/s and can be the source of Tasks 1-4; all tasks have the same size. In this example, the optimal schedule should let A be the source of Tasks 1 and 2, and B be the source of Tasks 3 and 4. However, if our greedy algorithm scans Task 1 first, it will assign it to node B, because B has more available bandwidth than A at this moment.

This observation suggests that, for a chunk which has a replica in an underemployed node, it's better to use the underemployed node as the source. To achieve this goal, we incorporate a distribution-driven prioritizing strategy: the *ObServer* first sorts all the nodes according to their available outgoing bandwidth in descending order, and sorts all the nodes according to their total sizes of common chunks in ascending order. Then, the *ObServer* picks the first β (5% in our typical settings) nodes from those two node lists respectively to form two sets, and gets the underemployed node set by computing the intersection of those two sets. Next, the *ObServer* selects all the recovery tasks that have replicas in the underemployed nodes, and puts them in a queue called "prioritized queue"; the *ObServer* puts the rest of the tasks in another queue called "normal queue".

We modify our greedy algorithm (§4.1) to incorporate this heuristic: the *ObServer* will first scan tasks in the prioritized queue and directly use the corresponding underemployed server as the source, instead of using the most under-utilized candidate. There are two corner cases: 1) it is possible that a prioritized task has replicas in more than one underemployed servers. In this case, the *ObServer* chooses the most under-utilized one among them; 2) though rare, it is possible that the underemployed server is saturated. In this case, the *ObServer* degrades the prioritized task into the normal queue, so that later we can still try its non-prioritized candidates.

Overhead. When searching underemployed nodes, Dayu maintains two heaps, whose keys are the available outgoing bandwidth and the total size of common chunks respectively, and whose values are the IDs of nodes. The *ObServer* will first build these two heaps, which is an $O(n)$ operation (n is the number of nodes) [23], and then pop 5% entries from these two heaps, with each pop an $O(\log n)$ operation. Our experiment shows that heapifying 10,000 entries and then popping up 5% of them only take a few miliseconds.

4.3 Allocating Bandwidth for Each Task

Given the source and destination of each recovery task, we need to answer how fast each task should proceed. A naive solution is to set a coarse-grained limit on all tasks within one node using $B_{recover_in/out}$ and let them compete for bandwidth. However, our experiments have revealed two problems with this approach: first, this approach may cause a congestion when the source's outgoing limit is larger than the destination's incoming limit. Although TCP can resolve such congestion eventually, it will cause packet drops and slow down recovery. Second, the competition may cause one task to be significantly slower than others, causing a significant straggler and violating our third goal.

Therefore, in this step, Dayu tries to set a constant rate for each task in one timeslot, with the goal of maximizing bandwidth utilization. Recall that we assume the NICs of the end hosts are the bottlenecks, so this step only considers the bandwidth utilization at the end hosts. Even so, this is still a challenging problem, since allocating bandwidth for a task will consume the bandwidth on both sides.

Dayu's solution is based on weighted shuffle scheduling (WSS) [14], a mature network scheduling algorithm designed for scheduling large data flows like data shuffle in MapReduce [24]. The key idea of WSS is that, to finish all the pairwise transfers at the same time, it guarantees that 1) transfer rates are proportional to data sizes for each transfer, and 2) at least one link is fully utilized. With WSS, only the bottleneck links (quite a minority) are fully used, while all the others have an amount of bandwidth left. In our scenario, however, WSS is not ideal: when considering unpredictable growth in foreground traffic, which may cause a non-bottleneck link to become a bottleneck in the middle of a timeslot, WSS may cause a waste of bandwidth, because Dayu could utilize more bandwidth of this link at the beginning.

To this end, Dayu introduces an iterative WSS solution to allocate bandwidth for each task. Its key idea is that, without delaying the bottleneck tasks, we should finish other tasks as early as possible, so as to reduce their completion time and to improve bandwidth utilization. Following this idea, if there is any remaining bandwidth after running one iteration of WSS, Dayu will use another iteration of WSS to identify the next bottleneck and allocate the remaining bandwidth.

To be specific, Dayu maintains a remaining incoming and outgoing bandwidth $B^i_{remain_in}$ and $B^i_{remain_out}$ for each node, whose initial values are $B^i_{recover_in}$ and $B^i_{recover_out}$. In each iteration, Dayu first finds the node with the longest $\frac{c^i_{in}}{B^i_{remain_in}}$ or $\frac{c^i_{out}}{B^i_{remain_out}}$, denoted T^*: the corresponding tasks are the bottlenecks. Then, Dayu allocates $B_t = \frac{s_t}{T^*}$ bandwidth to each task t, indicating that to minimize the completion time, the bottleneck tasks must be assigned a weighted fair share of the bandwidth, such that the weight of the share is proportional to s_t. Afterward, Dayu updates the remaining bandwidth as

$B^i_{remain_in} - = \frac{c^i_{in}}{T^*}$ and $B^i_{remain_out} - = \frac{c^i_{out}}{T^*}$ for each node i, removes the bottleneck tasks from their corresponding nodes, and updates the $c_{in/out}$ values of these nodes. Then Dayu moves to the next iteration with the remaining tasks, till there are no tasks remaining or the remaining tasks have an acceptable transmission time (i.e., less than or equal to the length of a timeslot) with their allocated bandwidth. Note that if a task goes through multiple iterations, its allocated bandwidth is the sum of the allocated bandwidth in each iteration.

Iterative WSS overcomes the drawbacks of WSS: since iterative WSS tries to allocate all bandwidth, it is not possible for the system to waste bandwidth when there are tasks that can utilize such bandwidth.

Our experiment shows that for a 3500-node cluster, each iteration will take at most 15 *ms*. Since dynamic convex hull node selection algorithm keeps the "$\frac{c}{B}$" values of most nodes to be close, the iterative WSS algorithm can usually finish within five iterations (i.e. 75 *ms*), which is acceptable.

4.4 Re-scheduling Straggling Tasks

Due to inaccurate workload estimation, sub-optimal scheduling, hardware exceptions, and etc., some tasks could not be finished at the end of one timeslot. Dayu has to re-schedule such straggler tasks in the next timeslot, but cannot simply treat them as new tasks, because changing the destination of one straggler task requires re-transmitting the task from the beginning, causing waste of bandwidth. Therefore, Dayu should avoid changing the destination when possible—this is a constraint new tasks do not have.

Identifying stragglers. Recall that Dayu overlaps different timeslots so that the scheduling phase of the current timeslot happens a short period of time (denoted as $T_{schedule}$) before the end of the last timeslot (Figure 2). Therefore, Dayu has to predict which tasks will become stragglers: for one unfinished task, Dayu uses its speed so far to estimate its speed till the end of the last timeslot; if the task cannot finish given the estimated speed, Dayu puts those tasks into a straggler set.

Prediction can certainly be inaccurate. If Dayu marks a task as a straggler but it actually finishes with the last timeslot, the corresponding nodes will simply ignore the new transmission plan scheduled by Dayu. Conversely, if a task is not marked as a straggler but it cannot finish within the last timeslot, the corresponding node will not get a new transmission plan, and thus will stick with the old plan. Both cases may cause inefficiency, but since $T_{schedule}$ is much smaller than $T_{timeslot}$, these two kinds of misidentification have little impact in our experiments.

Scheduling stragglers. First, Dayu will check whether the straggler set itself will saturate some nodes in the current timeslot. If any, the *ObServer* iteratively evicts the least finished task from each saturated node until it is no longer saturated. Those evicted tasks are categorized into two groups: tasks evicted from their sources and tasks evicted from their destinations. They are rescheduled in different manners.

- For each straggler task in the first group, the *ObServer* chooses a source and a transfer rate (the same as a new task), while keeping the destination unchanged, which means the task can resume from its current progress.
- For each straggler task in the second group, the *ObServer* reschedules it as a new task.

For unevicted straggler tasks, Dayu keeps their sources and destinations unchanged, and allocates the bandwidth with the iterative WSS algorithm. They can resume from their current progress.

Compared to treating stragglers as new tasks, Dayu tries to minimize re-transmitting data, since it only changes the destination of the second group of stragglers (quite a minority) and re-transmits their data. Compared to letting stragglers continue with their original plans, our experiments show that the introduction of straggler adjustment improves the overall recovery speed by 15.6%.

It should also be noted that how to detect and report slow hardware is an orthogonal problem [25–27]. Dayu assumes the system has some mechanism to measure and report the actual bandwidth of each node.

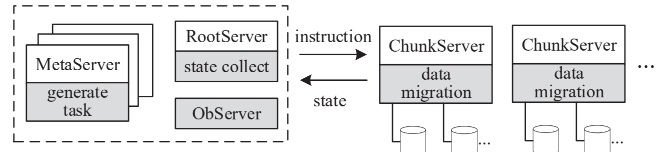


Figure 4: Our implementation of Dayu on Pangu. Gray boxes stand for functions or components Dayu adds to Pangu.

5 Evaluation

Our evaluation tries to answer the following questions:

1. How fast can Dayu complete one typical full node recovery and how much interference does Dayu introduce between background and foreground? (§5.1)
2. Could Dayu scale to even larger systems? (§5.2)
3. In Dayu, how much benefit does each key technology bring? (§5.3)
4. How does the setting of the parameters affect the performance of Dayu? (§5.4)

Implemention. We implement Dayu upon Pangu, by modifying *MetaServers*, *RootServer*, and *ChunkServers* of Pangu and introducing Dayu's *ObServer* into Pangu, as shown in Figure 4. The *ObServer* is aware of the information of all the recovery tasks as well as the global information provided by the *RootServer*, such as the chunk placement and the network utilization at each *ChunkServer*. As Pangu monitors and reports the states of *ChunkServers* every 15 seconds, the

timeslot length of this implementation is set to 15 seconds. Upon detection of a node failure in Pangu, *MetaServers* report all the data chunks of the failed node to the *ObServer*, which schedules how to re-replicate these data chunks. Afterwards, the *ObServer* instructs the *ChunkServers* to execute recovery tasks (i.e. re-replicate data chunks). Finally, after a recovery task is completed, the *ObServer* updates the *MetaServers* to reflect the locations of the newly-re-replicated chunk.

Testbed. We have deployed the Pangu-based Dayu implementation on a 1000-node cluster. Each node has two 12-core Intel E5-2630 processors, 96GB DDR4 memory, two 10Gbps NICs, 10 or 11 2TB hard disks, and Linux 3.10.0. Since our traces are collected from a cluster with 1Gbps NICs but our testbed is equipped with 10Gbps NICs, we add a traffic control to our testbed so that each NIC can only use 1Gbps bandwidth.

We have also built a simulation environment to test Dayu with the scale of 3,500 nodes or more. We run the simulation in a server with two 16-core Intel E5-2620 processors, 64GB DDR4 memory, and Linux 3.10.0.

Methodology. For experiments on real-world systems, we trigger data recovery by shutting down one *ChunkServer*. When performing recovery, we replay the trace collected from the real-world cluster system (§2.2). Since our testing cluster is smaller than the cluster where the trace is from, we reshape the trace to fit the cluster size by trimming or redirecting some requests, while keeping the ratio of read and write, the pressure on each node, and the degree of imbalance among nodes [28]. We record both the recovery time and the interference between the foreground and recovery traffic, which is measured by comparing the p90 latency (i.e., tail latency at 90th percentile) of the foreground requests with and without recovery traffic.

In the simulation experiments, we simulate the failure of a *ChunkServer* by sending its chunk information to Dayu. Since we do not actually run the system, we need to simulate the interference between the foreground and the recovery traffic. Due to the scale of the system, request level simulation takes very long, so we use flow level simulation as in [29, 30]. It simulates the bandwidth utilization of each link and periodically updates the utilization according to the foreground and recovery traffic information. We define the interference factor as the ratio between the overload traffic size and the link bandwidth, as follows:

$$B^i_{overload} = max(B^i_{recover} + B^i_{foreground} - 75\% \times B^i_{total}, 0) \quad (3)$$

$$F_{interference} = \frac{\sum_{i \in Nodes} B^i_{overload}}{\sum_{i \in Nodes} B^i_{total}} \quad (4)$$

The reason we define such an interference factor is that if the total bandwidth utilization exceeds 75% of the NIC's bandwidth, the foreground latency will increase significantly. To quantitatively understand this simulated interference factor, we map them to the p90 latency in the real-world experiments (§5.4): the short conclusion is that an interference factor smaller than 2% indicates very small interference and a factor close to or larger than 6.5% indicates very large interference. In our simulation experiments, we simulate 50 failure cases by randomly choosing 50 pairs of failed nodes and their failure time. For each algorithm, we simulate its performance on all the 50 cases and report its average performance numbers.

In our following experiments, Figure 5 presents the results from the real-world systems and the other figures present the results from the simulation experiments.

Comparison. In the experiments on the real-world systems, we compare Dayu with Pangu's original re-replication strategy, which adopts disk utilization aware random data placement and static rate control. We use three configurations Pangu-slow (limit recovery traffic to 30MB/s, which is the default configuration in production systems), Pangu-mid (90MB/s), Pangu-fast (150MB/s) as the baselines.

In the simulation experiments, we compare Dayu with different scheduling algorithms used in state-of-the-art systems (Table 2), with the exception of MCMF since its optimized solver is not open sourced. For fairness, we keep the node prioritizing and straggler adjustment part of Dayu, and plug in different node selection and bandwidth allocation algorithms. Specifically, when selecting the destination of recovery tasks, we compare Dayu's bucket dynamic convex hull algorithm (C) to the following algorithms: 1) *Random* (R), which randomly selects a node as the transmission source and destination; 2) *Best-of-two-random* (B2R), which first chooses two *ChunkServers* randomly, and then picks the lighter-loaded one as the source or destination [4, 9]; 3) *Weighted random* (WR), which uses the available bandwidth as the weight to randomly select a node; 4) *Greedy1* (G1), which scans all candidate *ChunkServers* as [31, 32], then in our scenario finds the one with minimal $\frac{c}{B}$. 5) *Greedy2* (G2), which chooses the lightest-loaded *ChunkServer*, by maintaining a red-black tree. All greedy algorithms, including Dayu, are executed using a single thread; all random-based algorithms are executed using 16 threads. Note that although random-based algorithms can be distributed to reach even higher speed, we find their speed is not the bottleneck anyway in our experiments. We also test the MILP algorithm with a state-of-the-art MILP solver Gurobi [33], but find it can only finish computation for a small-scale cluster; for a 3500-node cluster and only 2000 tasks, it cannot finish computation after 125 seconds and thus we do not report its results. When determining the rate of each task, we compare Dayu's iterative WSS (W) with deadline-based allocation (DA), which assigns a rate of $\frac{s_t}{T_{timeslot}}$ to task t so that a task can be finished in one timeslot.

5.1 Overall Performance

Evaluation on the Real-world Systems. Figure 5 shows the recovery times and the p90 latency of the foreground re-

System	Algorithm
Commons [3, 5–7]	Random placement
RAMCloud [4]	Best-of-two-random
CAR [31]	Greedy1
PPR [32]	Greedy1
Mirador [34]	Greedy2
DH-HDFS [35]	MILP
Sparrow [36]	Best-of-two-random
Firmament [37]	MCMF

Table 2: State-of-the-art systems and their algorithms.

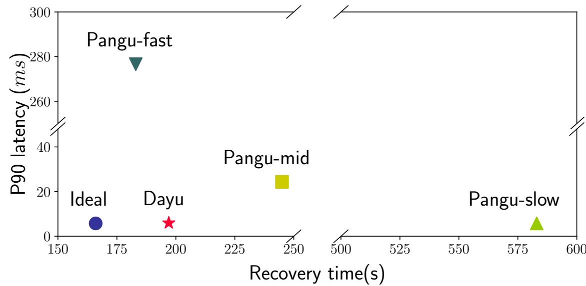


Figure 5: The recovery time and the p90 latency during recovery in real-world experiments

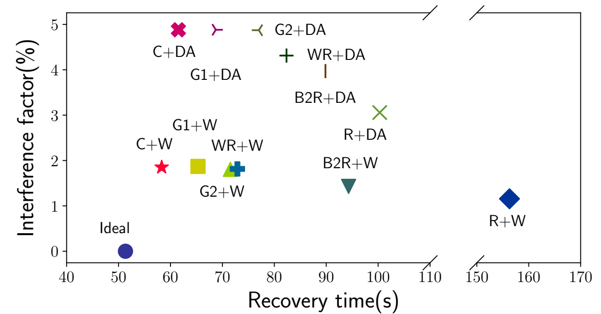


Figure 6: Data recovery in simulation (Dayu uses C+W)

quests during recovery. In this test, we shutdown a server to create 15TB of data to recover, and approximately 990 surviving *ChunkServers* are responsible for recovery. For comparison, we add an "Ideal" entry in Figure 5, which estimates the optimal recovery time assuming all available bandwidth ($\alpha = 75\%$) can be utilized, and introduces no interference on the foreground traffic (i.e., the foreground latency is identical to the one without recovery).

As shown in the figure, Dayu achieves near-optimal recovery speed as well as low interference. First, Dayu is approaching the ideal recovery speed, as its recovery time is 1.19× longer than "Ideal": this is 2.96× and 1.24× faster than Pangu-slow (default) and Pangu-mid configurations respectively. Compared with the Pangu-fast configuration, although Dayu has a slightly slower recovery speed (0.93×), it introduces far less interference on the foreground traffic. Considering the interference of the recovery traffic on the foreground traffic, Dayu's p90 latency is only 1.04× longer than "Ideal". Pangu-slow has a slightly lower interference with its p90 latency nearly the same as "Ideal"; Pangu-mid and Pangu-fast create unacceptable interference as their p90 latencies are 4.23-48.14× higher than "Ideal". Due to the high interference to the foreground traffic, Pangu-mid and Pangu-fast are seldom used on production clusters. In summary, compared with the different settings in Pangu, Dayu achieves close-to-optimal recovery time and interference.

Evaluation on the Simulation Systems. Figure 6 shows the results of simulation experiments. Again, compared with other algorithms, Dayu achieves a good balance between recovery speed and interference.

In terms of recovery speed, Dayu's combination of dynamic convex hull node selection and iterative WSS (C+W) can achieve the shortest recovery time among all algorithms, which is 1.14× longer than the ideal recovery time. Compared to other algorithms, dynamic convex hull node selection achieves the fastest recovery speed, which is 1.12× faster than G1, the second best one. Note that though G1 is close to Dayu in this experiment, it does not scale well due to its high computation overhead (§5.2). For the greedy-based algorithms including Dayu, iterative WSS is slightly faster than deadline-based allocation, because the former can finish the last timeslot early when tasks are rare, while the latter must finish tasks at the end of the last timeslot. For random-based algorithms, such effect is unclear because the recovery speed is mainly determined by the selection of the sources and destinations.

In terms of interference on foreground, Dayu has acceptable interference factor (recall that a factor of 2% is small and a factor larger than 6.5% is unacceptable). With the same bandwidth allocation strategy, Dayu's node selection algorithm and other greedy algorithms have slightly larger interference than those random based algorithms, because greedy algorithms usually utilize more estimated available bandwidth. When the estimation of the foreground traffic has some errors, the interference will be slightly larger. With the same node selection algorithm, iterative WSS consistently brings lower interference than deadline-based allocation (DA).

5.2 Scalability

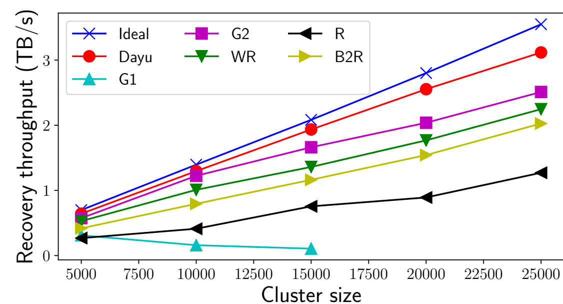


Figure 7: Dayu's scalability

We evaluate the scalability of Dayu beyond 3,500 nodes. To measure the full capability of different algorithms, we assume there are infinite number of recovery tasks and simulate how much data each algorithm can recover in 20 timeslots. As the scale of the simulated clusters are larger than our observed cluster, we randomly generate block placement based on the statistics from our collected traces; we randomly pick the foreground trace from one real node for one simulated node.

As shown in Figure 7, Dayu can scale to 25,000 nodes and till that point, the performance of Dayu is higher than all other algorithms. We do not test even larger scales because they are too far away from our target (10K nodes). Besides Dayu, all random algorithms scale pretty well, which is as expected, though their performance is not as good as Dayu. G1 does not scale to more than 5,000 nodes because of its high computation overhead. Note that as greedy algorithms, Dayu and G2 will eventually stop scaling at some point because of their centralized computation, but at least for the scale we target now and in the near future, the simulation shows that Dayu is fast enough and can provide better quality.

5.3 Effects of Individual Techniques

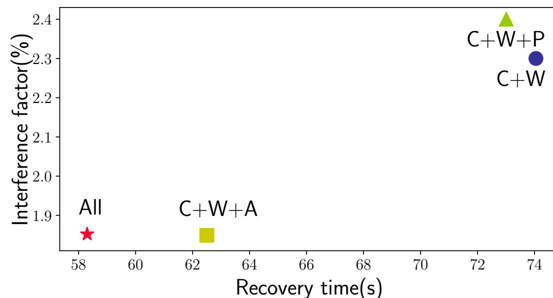


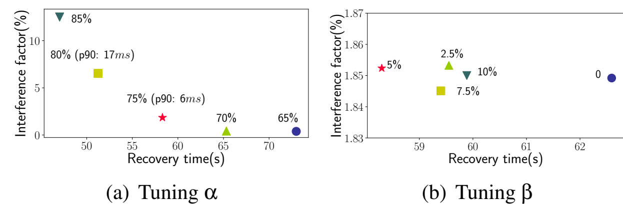
Figure 8: Effects of prioritizing underemployed node (P) and re-scheduling stragglers (A)

We further investigate the effects of prioritizing underemployed nodes (P) and re-scheduling stragglers (A) described in Section 4.2 and 4.4. We use Dayu equipped with convex hull node selection (C) and iterative WSS bandwidth allocation (W) as the baseline (C+W), which scans tasks with no prioritization and executes stragglers with the original plan (i.e. P and A are disabled). Note that in this baseline, Dayu is aware of those stragglers and will use their information to schedule the current timeslot but won't re-schedule stragglers.

Figure 8 presents Dayu's schedule results with and without P and A. As shown in the figure, the re-scheduling of stragglers is keen to the performance: compared with the baseline (C+W), re-scheduling stragglers (C+W+A) reduces recovery time by 15.6% and reduces the interference as well. Though prioritizing underemployed nodes has limited effect without re-scheduling stragglers, it accelerates the recovery speed by 7.2% when straggler re-scheduling is already equipped (compare "All" to the case C+W+A).

5.4 Impacts of Key Parameters

Finally, we measure the impacts of key parameters of Dayu. The first one is α in Equation 1, which controls the interference of recovery traffic on foreground traffic. Figure 9(a) plots the recovery time and interference factor as α increases from 65% to 85% with the step size of 5%. One can see that the larger the α, the shorter the recovery time but the larger the interference factor. In this figure, we further map some of these simulated interference factors to the p90 latencies from the real-world experiments, so that we can quantitatively understand the values of the simulated interference factors. Our decision to use the value 75% for α is mainly based on these p90 latencies from real-world experiments: with $\alpha = 75\%$, Dayu achieves close-to-optimal recovery time and p90 latency (§5.1); with $\alpha = 80\%$, although Dayu decreases recovery time by 9.1%, it almost triples the p90 latency of the foreground traffic.



(a) Tuning α (b) Tuning β

Figure 9: Effects of tuning α and β

The next parameter, β, denotes the ratio of selected nodes from two sorted lists when choosing underemployed *ChunkServers* in Section 4.2. We change β from 0% to 10% with the step size of 2.5%. As shown in Figure 9(b), the value of β has no significant impact on the interference factor and setting it to 5% achieves the lowest recovery time, which is why Dayu sets β to 5%.

Another important parameter is the length of a timeslot ($T_{timeslot}$), but since this parameter affects the overall overhead of Pangu, we were not allowed to change it in the production system and thus we were not able to record and analyze a trace with a different $T_{timeslot}$. In general, shorter $T_{timeslot}$ will benefit Dayu by allowing it to react to foreground fluctuation more quickly but will increase the overhead of Pangu.

6 Related Work

Data Recovery. Popular distributed filesystems such as GFS [3], HDFS [5], Cosmos [6], and Windows Azure Storage [7] use random node selection and static rate control for data recovery, same as Pangu. RAMCloud [4, 38] uses the best-of-two-random algorithm to select the source and destination for a recovery task. Constrained by the deterministic placement, consistent hashing based storage systems [17, 22, 39–41] have little flexibility to choose the destination. Our work shows that randomized algorithms may not have a good quality in a highly imbalanced environment.

Some works improve data recovery in erasure-coded storage, by accelerating recovery of one failed block [31, 42, 43] or designing new recovery-efficient codes [44–47]. Applying Dayu to erasure-coded storage is our future work.

Data migration. Data recovery can be viewed as a subtopic of data migration. A number of distributed filesystems [3, 5] trigger data migration with a simple strategy or even manually (e.g., running HDFS balancer [48]). Mirador [34] uses a priority queue to greedily migrate data objects according to pre-defined rules. However, experiments in [34] report it does not scale well due to its greedy algorithm. Curator [49] uses a reinforcement learning solution to determine when to start a migration task, but it does not choose sources and destinations for data migration. DH-HDFS [35] utilizes MILP solver to manage migration of large scale storage system, but for our problem, MILP is too slow.

Constrained data placement strategies. Besides consistent hashing based storage systems [17, 22, 39–41], there are other systems restricting the data placement. Facebook [50] modifies native HDFS to constrain the placement of block replicas into smaller node groups (i.e., with a smaller scatter width), reducing the probability of losing data due to simultaneous node failures. With a fixed scatter width, CopySets [51] and Tiered Replication [52] further try to minimize the number of the distinct copysets in the whole system to reduce the probability of data loss due to correlated node failures. We plan to investigate the applicability and effectiveness of Dayu on these strategies in the future.

Large scale scheduling. Many large-scale computation platforms need to schedule computation tasks, which is similar to schedule recovery tasks in Dayu. Most centralized schedulers [53, 54] have poor computation performance at a large scale, and thus distributed schedulers are widely discussed [36, 55, 56]. However, due to the lack of coordination and the latest state, these schedulers often fail to generate high quality decisions [37]. Firmament [37], a centralized scheduler, succeeds to scale to a 12500-node cluster [57] , but experiments in [37, 58] report it has limited scalability with massive short tasks, which is exactly our scenario (§2.2).

7 Conclusion

Our work shows that a centralized scheduler has better scheduling quality, especially in a dynamic and imbalanced environment; its weakness, i.e. relatively low speed compared to the decentralized schedulers, can be mitigated by different optimizations (e.g. timeslot-based scheduling, convex hull optimization, etc). As a result, it can support a reasonably large system we target.

8 Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd Sudarsun Kannan for his guidance during our camera-ready preparation. We also thank Tianyang Jiang for helpful discussions, and Alibaba Cloud for providing us the evaluation cluster. This work was supported by the National key R&D Program of China under Grant No. 2018YFB0203902, and the National Natural Science Foundation of China under Grant No. 61672315.

References

[1] Zujie Ren, Weisong Shi, Jian Wan, Feng Cao, and Jiangbin Lin. Realistic and scalable benchmarking cloud file systems: Practices and lessons from alicloud. *IEEE Transactions on Parallel and Distributed Systems*, 28(3272-3285):1, 2017.

[2] Backblaze. Backblaze Durability is 99.999999999% — And Why It Doesn't Matter. https://www.backblaze.com/blog/cloud-storage-durability/, 2018. Online; accessed 2018-12-25.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, volume 37. ACM, 2003.

[4] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 29–41. ACM, 2011.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of 26th IEEE symposium on Mass storage systems and technologies (MSST'10)*, pages 1–10. IEEE, 2010.

[6] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment (VLDB'08)*, 1(2):1265–1276, 2008.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 143–157. ACM, 2011.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[10] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. BDS: a centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth European Conference on Computer Systems (Eurosys'18)*, pages 10:1–10:14. ACM, 2018.

[11] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3Sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth European Conference on Computer Systems (Eurosys'18)*, page 2. ACM, 2018.

[12] 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)*, page 35. ACM, 2016.

[13] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, 2008.

[14] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'11)*, pages 98–109, New York, NY, USA, 2011. ACM.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[16] Apache. HDFS Federation. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html, 2018. Online; accessed 2018-04-16.

[17] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.

[18] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)*, volume 39, pages 51–62. ACM, 2009.

[19] Albert Greenberg, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO'08)*, pages 57–62. ACM, 2008.

[20] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)*, volume 39, pages 39–50. ACM, 2009.

[21] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'08)*, volume 38, pages 63–74. ACM, 2008.

[22] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, volume 41, pages 205–220. ACM, 2007.

[23] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04)*, 51(1):107–113, 2004.

[25] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 1–14, Oakland, CA, 2018. USENIX Association.

[26] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[27] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in Situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 1–16, Carlsbad, CA, 2018. USENIX Association.

[28] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'13)*, volume 43, pages 231–242. ACM, 2013.

[29] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, Berkeley, CA, USA, 2010. USENIX Association.

[30] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'12)*, pages 127–138. ACM, 2012.

[31] Zhirong Shen, Jiwu Shu, and Patrick PC Lee. Reconsidering single failure recovery in clustered file systems. In *Proceedings of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 323–334. IEEE, 2016.

[32] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, page 30. ACM, 2016.

[33] Gurobi. Gurobi 8.0. URL:http://www.gurobi.com, 2018. Online; accessed 2018-12-25.

[34] Jake Wires and Andrew Warfield. Mirador: An active control plane for datacenter storage. In *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 213–228, Santa Clara, CA, 2017. USENIX Association.

[35] Pulkit A. Misra, Inigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *Proceedings of 2017 USENIX Annual Technical Conference (ATC'17)*, pages 799–811, Santa Clara, CA, 2017. USENIX Association.

[36] 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, pages 69–84, New York, NY, USA, 2013. ACM.

[37] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 99–115, Savannah, GA, 2016. USENIX Association.

[38] Ryan Scott Stutsman. *Durability and crash recovery in distributed in-memory storage systems*. PhD thesis, Stanford University, 2013.

[39] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[40] Gluster. GlusterFS documention. URL:https://docs.gluster.org/en/latest/, 2018. Online; accessed 2018-12-25.

[41] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the ACM SIGCOMM 2001 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'01)*, 31(4):149–160, 2001.

[42] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*, pages 567–579, 2017.

[43] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, page 30. ACM, 2016.

[44] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[45] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 81–94, 2015.

[46] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST'18)*, volume 2018, pages 139–154, 2018.

[47] Min Ye and Alexander Barg. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory*, 63(10):6307–6317, 2017.

[48] Apache. HDFS Balancer Command. URL:https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html, 2019. Online; accessed 2019-01-10.

[49] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 51–66, Boston, MA, 2017. USENIX Association.

[50] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*, pages 1071–1080. ACM, 2011.

[51] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 37–48, 2013.

[52] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 31–43, 2015.

[53] 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 (SOSP'09)*, pages 261–276. ACM, 2009.

[54] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[55] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[56] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, pages 499–510, Santa Clara, CA, 2015. USENIX Association.

[57] 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)*, page 7. ACM, 2012.

[58] Ionel Corneliu Gog. *Flexible and efficient computation in large data centres*. PhD thesis, University of Cambridge, 2018.

OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface

Yun-Sheng Chang and Ren-Shuo Liu
Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan

Abstract

Consumer-grade solid-state drives (SSDs) guarantee very few things upon a crash. Lacking a strong disk-level crash guarantee forces programmers to equip applications and filesystems with safety nets using redundant writes and flushes, which in turn degrade the overall system performance. Although some prior works propose transactional SSDs with revolutionized disk interfaces to offer strong crash guarantees, adopting transactional SSDs inevitably incurs dramatic software stack changes. Therefore, most consumer-grade SSDs still keep using the standard block device interface.

This paper addresses the above issues by breaking the impression that increasing SSDs' crash guarantees are typically available at the cost of altering the standard block device interface. We propose Order-Preserving Translation and Recovery (OPTR), a collection of novel flash translation layer (FTL) and crash recovery techniques that are realized internal to block-interface SSDs to endow the SSDs with *strong request-level crash guarantees* defined as follows: 1) A write request is not made durable unless all its prior write requests are durable. 2) Each write request is atomic. 3) All write requests prior to a flush are guaranteed durable. We have realized OPTR in real SSD hardware and optimized applications and filesystems (SQLite and Ext4) to demonstrate OPTR's benefits. Experimental results show 1.27× (only Ext4 is optimized), 2.85× (both Ext4 and SQLite are optimized), and 6.03× (an OPTR-enabled no-barrier mode) performance improvement.

1 Introduction

Storage systems are constructed layerwise; thus the overall system performance depends on an appropriate division of labor between layers. For example, for applications and filesystems that run on top of flash-based solid-state drives (SSDs), if the underlying SSDs focus too much on optimizing their own performance and maintain too weak guarantees upon a crash or power loss (crash for short), programmers are forced to equip the applications and filesystems with safety nets using redundant writes and flushes [24, 26, 33, 41], which in turn complicate the overall system and degrade the overall performance.

That being said, widely used, consumer-grade SSDs (baseline SSDs for short) guarantee very few things upon a crash:

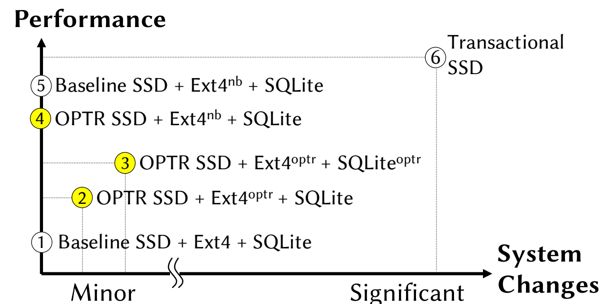


Figure 1: OPTR SSD vs baseline and transactional SSDs. Ext4^{optr} and SQLite^{optr} denote our optimized versions of Ext4 and SQLite, respectively. Ext4^{nb} denotes Ext4 mounted with the -o nobarrier option.

each individual sector that is written since the last flush may either be done or left undone, which means there is an enormous set of post-crash states for applications and filesystems to handle. Lacking a strong disk-level crash guarantee has caused deep, long-term consequences. For example, the ordering constraint among user data, metadata, journal, and commit records should be enforced to prevent sensitive user data and inconsistent states from being accidentally exposed. However, disks are not obligated to preserve any write order, and filesystems are reluctant to use flushes to enforce a write order due to severe performance penalties. Therefore, Linux Ext3 even turned off flushes for many years at the risk of crash vulnerabilities [20]. Another alarming example is that since the crash guarantees of underlying disks are weak, various filesystems struggle to provide a standardized and strong crash guarantee at their level, and this struggle consequently makes applications have difficulty ensuring correct recovery from a crash. For example, LevelDB and Git were recently found to contain many crash vulnerabilities owing to this reason [39].

Despite many issues, SSDs with a weak crash guarantee are still widely used for two reasons. First, the weak crash guarantee has existed since the hard disk drive (HDD) era. This design choice was unavoidable for optimizing HDD performance and remains as a matter of course for optimizing SSD performance. For example, HDDs were permitted to freely reorder requests according to the position of pickup

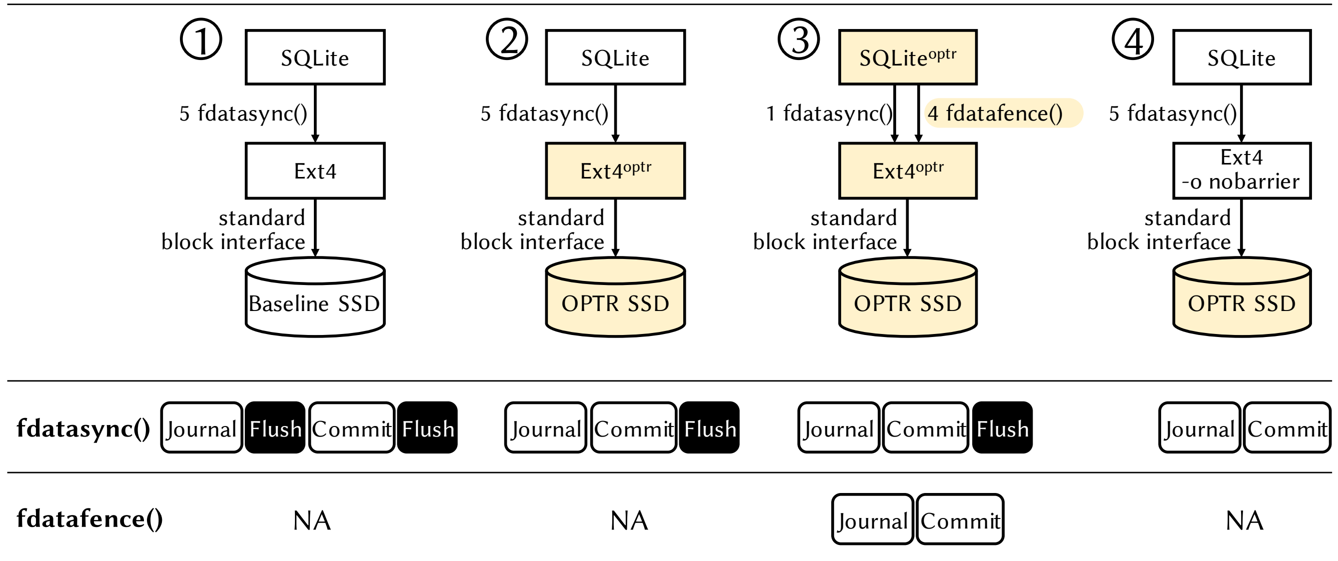


Figure 2: Usage modes of OPTR SSDs and their filesystem and disk interfaces. *fdatafence* is our newly proposed filesystem primitive that incurs no flush. More details on the original five *fdatasync* and the use of *fdatasync* are in Section 4.

heads to minimize seek and rotational latency, and thus, SSDs are also allowed to reorder or parallelize requests to maximize their internal channel- and chip-level parallelism. The second reason is that although novel SSDs with revolutionized interfaces that offer transaction-level crash guarantees (transactional SSDs for short) [28, 32, 37, 40, 44] have recently been studied, the fact is that there has been a vast body of software and systems developed based on a *standard block device interface* (block interface for short, e.g., SATA). Therefore, as shown in Figure 1, although transactional SSDs (⑥) can potentially offer much higher performance than baseline SSDs (①), only few systems can adopt such dramatic changes. Not surprisingly, most consumer-grade SSDs keep using the block interface, and thus, the benefits of transactional SSDs are not widely available.

This paper addresses the above issues by breaking the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the block interface. We propose Order-Preserving Translation and Recovery (OPTR), a collection of novel flash translation layer (FTL) and crash recovery techniques that are realized internal to a block-interface SSD to endow the SSD with ***strong request-level crash guarantees*** as defined as follows:

1. **Prefix Semantics**: The SSD does not make a write request durable unless all the write requests received previously by the SSD are durable (stronger than baseline SSDs).
2. **Request Atomicity**: Each write request received by the SSD is atomic regardless of the request size (i.e., number of sectors) (stronger than baseline SSDs).
3. **Flush Semantics**: The SSD guarantees durability to all write requests that are received prior to a flush (identical to baseline SSDs).

Figure 3 uses a four-sector SSD to demonstrate the *strong request-level crash guarantees*. The four sectors initially store four version numbers, 0, 0, 0, and 0, respectively. The SSD receives four write requests and one flush request before a crash occurs. Each write request is specified by its *lba* and *size* in parentheses. We assume that write requests always increment the version number(s) of the written sector(s). For example, the first write request touches the first and second sectors, and thus the four version numbers become 1, 1, 0, and 0. As shown in the figure, each sector of a baseline SSD can exhibit two to three valid post-crash version numbers; therefore, the baseline SSD can exhibit $2\times2\times3\times3=36$ valid post-crash results. In comparison, an OPTR SSD guarantees to complete write requests in order and atomically, and there are only two write requests after the last flush; therefore, the number of valid results is significantly confined to three.

The *strong request-level crash guarantees* and our intentional choice to use a *standard block interface* bring several benefits. First, for programmers who want to develop new applications or filesystems, since the crash guarantees of OPTR SSDs are truly intuitive, the chance of making mistakes decreases. Additionally, since OPTR significantly confines the number of valid post-crash states, testing or verifying the correctness of a program becomes more manageable. Second, in addition to developing new programs, it is also simple for programmers to optimize existing applications and filesystems to benefit from OPTR. For example, Ext4 and SQLite resort to flushes to realize barriers because barriers are unavailable to most SSDs [45]. We refer to these flushes as ***unnecessary flushes***. With OPTR's *strong request-level crash guarantees*, we can optimize Ext4 and SQLite to remove

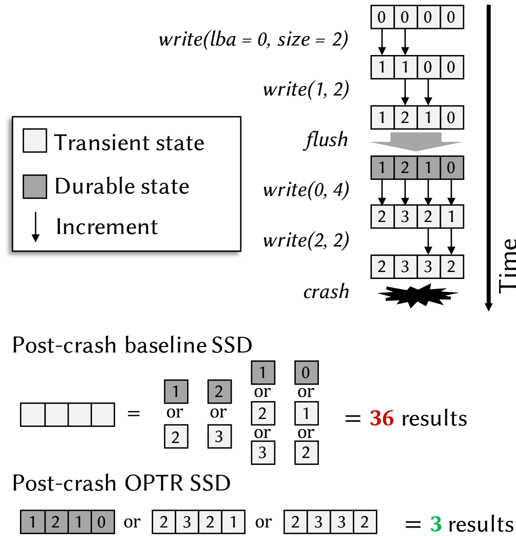


Figure 3: Post-crash states of OPTR SSDs vs. those of baseline SSDs.

unnecessary flushes (Section 4). The changes made to SQLite and Ext4 code are minor, and the achieved performance is significant, as illustrated by ② (optimizing Ext4 only) and ③ (optimizing both Ext4 and SQLite) in Figures 1 and 2. Finally, all existing, unmodified applications and filesystems can still run on OPTR SSDs, and this backward compatibility enables an *OPTR-enabled no-barrier mode* as described as follows.

In the ***OPTR-enabled no-barrier mode***, a filesystem is mounted with a no-barrier option (e.g., -o nobarrier for Ext4) and run on top of an OPTR SSD, as illustrated by ④ in Figures 1 and 2. In contrast to mounting the filesystem with the no-barrier option on a baseline SSD (⑤ in Figure 1), this mode achieves the best of both worlds, i.e., high performance (which is the reason why Ext3 used to disable flushes [20]) and consistency guarantees (because OPTR preserves order without the need for explicit flushes). This mode is practical and useful for smartphones, consumer-grade computers, and less-critical database systems such as SQLite [3] and some key-value stores [18][1].

This work makes the following contributions.

- This is the first SSD work with *strong request-level crash guarantees* and a standard block interface. We change the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the standard block device interface. We present the address translation, garbage collection (GC), and crash recovery algorithms internal to SSDs to achieve OPTR.
- We extend and restructure the FTL of an academic domain SSD project (OpenSSD [2]) to equip it with a write cache as our baseline. This FTL implementation is more sophisticated and modularized.
- We extend the FTL mentioned above to realize OPTR.
- We develop a functional simulator that can simulate an FTL given IO requests and crash events at high speed to test whether the FTL can recover from a crash and meet the OPTR requirement. We have validated this simulator and our implemented FTLs against each other. We anticipate that the simulator itself will be a useful tool for future FTL research.
- We exploit the *strong request-level crash guarantees* by optimizing the *fsync* primitive of Ext4 and newly proposing an *ffence* primitive for Ext4. These two primitives help to eliminate *unnecessary flushes*. We also demonstrate optimizing a database system, SQLite, to exploit the two primitives.

The rest of this paper is organized as follows. Section 2 describes the background of SSDs. Section 3 presents the detailed design and implementation of OPTR. Section 4 demonstrates the filesystem and application optimizations enabled by OPTR. Section 5 shows the results of validating OPTR using our developed functional simulator. Section 6 evaluates OPTR's gain and overhead. Section 7 analyzes the FTL in more detail. Section 8 surveys related work, and Section 9 concludes this work.

2 Background

2.1 Flash Translation Layers

In the core of an SSD lies an FTL. FTLs are responsible for translating a sequence of host-issued requests, including *write*, *read* and *flush*, into a sequence of flash operations, including *page-program*, *page-read*, and *block-erase*. Both read and write requests specify the address range of the data at the sector granularity; flush instructs an SSD not to acknowledge the host until the SSD persists all cached writes on stable media.

Upon receiving a write request, the FTL segments the write data into pages based on the given address range. A write request involves modifications to the logical-to-physical (L2P) mapping table. Many mapping schemes have been proposed (e.g., page-level, block-level, and hybrid) for various cost-performance tradeoffs. This work is based on page-level FTLs. Page-level FTLs divide the logical address space of SSDs into pages, which are indexed by a logical page number (LPN). Each entry in the L2P table maps an LPN to the location of a flash page indexed by a physical page number (PPN).

To handle a read request, the FTL translates the address range into LPNs and performs L2P table lookups to obtain the PPNs of the requested pages. If the cache happens to have a requested page, the contents are returned to the host; otherwise, a page-read is performed to retrieve the contents from the flash memory.

Flush requests are synchronous; thus, upon receiving a flush, the FTL must persist all dirty data in the write cache

[1]Note that this mode is not suitable for critical systems such as financial transaction processing systems because it relaxes durability.

and ensure no ongoing page-program operation exists before returning a success acknowledgment to the host.

As flash memory forbids in-place updates, overwriting data is done by writing the updated data to a free page and leaving the outdated data in the original page. A dedicated routine, called GC, is designed to reclaim these invalid pages, which store outdated data. The GC routine starts with choosing a victim block based on, e.g., the greedy policy [29, 42]. Then, this routine performs a series of page-reads and page-programs to relocate the valid data in the victim block. Finally, the victim block is erased and added to the free-block list.

2.2 High-Performance Schemes

Modern SSDs achieve significant performance mainly owing to the following three schemes: internal parallelism, request scheduling, and write caching. Note that these high-performance schemes all break the write order.

SSDs typically consist of multiple independent internal channels for transferring data and commands. Each channel connects to multiple flash chips. The **internal parallelism** of SSDs comes from these channels and chips, which can perform flash operations independently.

The goals of **request scheduling** are to increase the number of parallelized flash operations and to decrease the latency of each request. The former can be achieved by reordering requests to reduce resource conflicts [34]. The latter can be achieved by prioritizing requests with lower latency [22, 27].

Write caching removes slow storage accesses from the critical path. As the access speed of flash memory is often orders of magnitude slower than that of DRAM, most commercial SSDs employ a DRAM-based write-back cache for better performance. Another advantage of write caching is write coalescing. Multiple writes targeting the same LPN have the opportunity to be merged into one page-program operation, resulting in higher throughput and longer flash lifetimes.

2.3 SSD Recovery

The recovery mechanism of SSDs rebuilds the L2P table after a system crash. Some previous works have studied SSD recovery [7, 8, 11, 17, 30]. Leaving LPN information in the spare area of each written page during writes is the most common strategy [7], and the L2P information can be reconstructed after a crash by scanning this information. As multiple physical pages may contain the same LPN, a sequence number is often used to determine their validity.

The process of rebuilding the L2P table may need to read the spare areas of an enormous number of pages, leading to long recovery times. For an SSD with 512 GB capacity, assuming that each page is 32 KB and reading 32 pages in parallel takes 100 μs, reading the spare area of all the flash pages can cost up to three minutes, which is unacceptable in most situations. Therefore, optimization to reduce the recovery time has been proposed. Birrell et al. store the abovementioned per page recovery information in the last page of a block when the block is fully written [8]. This design eliminates the need to read an entire block. Bjorling et al. take partial and full checkpoints of the L2P table and persist the images in a reserved area [9]. During recovery, these images are loaded as the initial L2P table, and pages written after the latest checkpoint are subsequently remapped.

3 OPTR SSD Design

We realize the following five components internal to an SSD to achieve *strong request-level crash guarantees* without changing the standard block interface: 1) tracking the completion of write requests to ensure request atomicity, 2) tracking the coalescing between write requests, 3) periodic mapping table checkpointing, 4) tracking the availability of blocks for GC, and 5) order-preserving recovery.

3.1 Write Completion Tracking

Request atomicity is one of OPTR's guarantees. To achieve this coarse-grained atomicity, OPTR determines the completion and incompletion of each individual write request using a simple strategy: a write request that involves N pages is completed if and only if those N pages do exist in flash after a crash.

More specifically, OPTR leaves the following three kinds of clues in the spare area of each written flash page to facilitate determining the completion of a write request afterward. The first is a unique sequence number of the write request (wid, an 8-byte integer) assigned by the FTL according to the order in which the request is received. The second is the size of the write request in number of pages ($size$, a 4-byte integer). The third is the logical page number (lpn, a 4-byte integer) of the written page.

To determine which writes are completed and which are not, the recovery procedure divides flash pages into groups according to their write request identifiers (wid), counts the appearance of each wid, and then compares this count with $size$ for each write request. The entire write request is completed if and only if the count matches the request size.

3.2 Write Coalescing Tracking

Two or more write requests may coalesce in the write cache of SSDs; we refer to the involved write requests as coalesced write requests. This situation reduces the in-flash appearance count of wid of the involved write requests.

OPTR allows write coalescing to happen instead of avoiding it. To this end, OPTR detects and records each coalesced page. Each page in the write cache is tagged with a dirty flag, a wid tag, and a $size$ tag. By doing so, whenever a dirty cache page is overwritten, OPTR detects that a coalesced page exists (and anticipates that the appearance of the corresponding wid in flash decrements). For each coalesced page, OPTR records (in a DRAM buffer) the request IDs of the two involved write requests and the size of the prior write request. For example, as shown in Figure 4, a coalescing record with

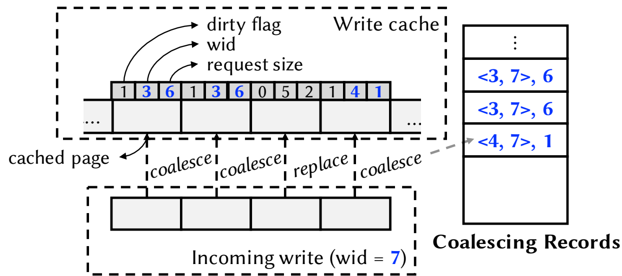


Figure 4: Write cache and generated coalescing records.

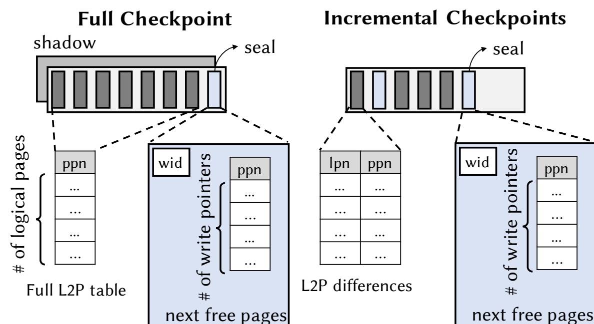


Figure 5: Full and incremental checkpoints.

"<3, 7>, 6" denotes that a prior write request whose ID is 3 coalesces with a later write request whose ID is 7, and the size of the prior write request is 6 pages. If a write involves multiple coalesced pages, multiple pieces of information are recorded. A batch of coalescing records are committed to flash when the buffer is full or when the SSD is requested to perform a flush (either issued externally by the host or internally by OPTR mechanisms).

It is hard to define the completion of each single write request that coalesces with other requests because the completion of multiple coalesced write requests needs to be atomic (which OPTR guarantees). In contrast, it is relatively simple to define incompletion as follows. Let P_i be the number of pages whose $wid = i$, D_i be the number of recorded $< x, y >$ pairs with $x = i$, and $Size_i$ be the size of the write request with $wid = i$. A coalesced write request with $wid = i$ is incomplete if $P_i + D_i < Size_i$.

3.3 Mapping Table Checkpointing

The L2P mapping table is checkpointed to flash to speed up recovery. More specifically, we reserve the first few flash blocks of each flash chip for checkpoints. OPTR keeps two types of checkpoints, a full checkpoint and several incremental checkpoints, as shown in Figure 5. The two types of checkpoints differ in that a full checkpoint snapshots the entire L2P table, while an incremental checkpoint records only the differences in the L2P table since the latest checkpoint (either full or incremental). In addition to the L2P table, both types of checkpoints record a seal page at the end of a checkpoint that includes 1) the latest wid at the time the checkpoint is made and 2) the PPNs of the next free flash pages at the time the checkpoint is made. OPTR employs incremental checkpoints by default. When the flash area for storing incremental checkpoints is full, OPTR creates a new full checkpoint and then clears the incremental checkpoints. OPTR employs a shadow for the full checkpoint to ensure its integrity, and the wid can be used to determine the recency between the full and incremental checkpoints after a crash. The PPNs of the next free flash pages allow L2P updates after the latest checkpoint to be retrievable. More specifically, for each PPN, subsequent written pages in the same block are available as pages within a block are sequentially written, and the summary page of the block can direct OPTR to the subsequent written block and so on. Then, the L2P updates can be derived from the LPN information in the spare area of these retrieved pages.

3.4 Garbage Collection

Flash pages that store outdated data are commonly considered invalid and useless, but it is these pages that OPTR leverages to rollback a disk from a crash to an order-preserved state. Thus, we enforce two constraints on the GC routine. The first constraint forbids GC from reclaiming pages programmed *after* the latest checkpoint. Since OPTR determines the completion of a write issued after the latest checkpoint by the number of pages owned by the write, reclaiming pages written prior to a flush but after the latest checkpoint would result in a flush semantics violation.

The second constraint forces an internal flush before performing GC. This flush ensures that every page being erased by GC has a stable counterpart that can always survive across a crash. To reduce the performance penalty of an internal flush, we amortize its cost by conducting GC to a batch of blocks (16 blocks in our case).

3.5 Order-Preserving Recovery

The recovery process is divided into the following phases: 1) recover the L2P table according to the latest full checkpoint, 2) sequentially incorporate the L2P differences recorded in the incremental checkpoints into the L2P table under recovery, 3) count the wid of flash pages written after the latest checkpoint according to the page pointers recorded in the latest checkpoint, and determine the completion/incompletion of each write request, 4) recover write requests after the latest checkpoint using a flow network, which we will describe shortly, and 5) sequentially incorporate the L2P changes of the write requests after the latest checkpoint.

The fourth phase above needs awareness of the order, completion, and coalescing of write requests to recover the disk to a state that satisfies our claimed guarantees. To this end, we formulate the task as a flow network problem as follows. As shown in Figure 6, write requests are represented by vertices. The order of immediately successive write requests is

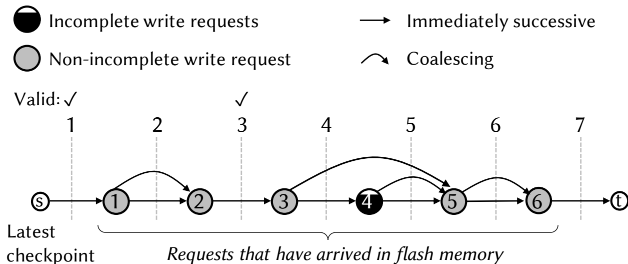


Figure 6: Determining the optimal recovery point by finding a cut in a flow network.

denoted by the straight directed edges pointing from request $wid = n$ to request wid=n+1, and coalescing between write requests is denoted by bent directed edges pointing from the earlier request to the later request. Each vertex is labeled by a wid and a Boolean value that denotes whether the write request is incomplete. Two additional vertices, s and t, are connected to the first and last vertices, respectively, to form a flow network from source (s) to sink (t), where s can be viewed as the latest checkpoint.

With such a formulation, finding a valid order-preserving recovery point is equivalent to finding an s-t cut such that 1) the cut size is equal to one (i.e., only a straight edge but not bent edges crosses the cut) and 2) the subgraph that contains vertex s (referred to as subgraph S for short) needs to contain no vertex denoting that the corresponding write request is incomplete. The rationales are as follows. The construction of the graph ensures that the write requests in subgraph S must preserve the order. A cut whose size is equal to one implies that coalesced write requests must be atomic. Since all write requests in subgraph S are not incomplete, it is valid for OPTR to recover them and drop others.

Let us take Figure 6 as an example. There are six write requests (1 to 6) present after the latest checkpoint, and some of their data pages have arrived in flash memory. The construction of the flow network reveals seven possible s-t cuts, 1 to 7, each representing a potential recovery point. Among the seven cuts, only 1 and 3 are valid. Cuts 5 to 7 are invalid recovery points because request 4 is incomplete, which breaks the order-preserving guarantee. Cuts 2 and 4 are invalid recovery points because they both cause coalesced requests to tear apart. Let us take Cut 4 as an example. Since some pages of request 3 are coalesced by request 5, request 3 cannot exist alone without request 5.

Note that OPTR can also handle multiple coalesces to the same page by creating multiple curved edges in the flow network. Again, take Figure 6 as an example. Consider a scenario where request 5 modifies a dirty cached page written by request 3, and then request 6 modifies the same page while this page is still dirty in the cache. This scenario would result in one curved edge from 3 to 5 and one from 5 to 6, as shown in Figure 6.

The optimal order-preserving recovery point should include as many write requests as possible. Therefore, finding this point is equivalent to finding the abovementioned s-t cut with the maximal subgraph S. A naive algorithm to find the optimal order-preserving recovery point is to start from a subgraph S containing only vertex s and gradually add vertices to subgraph S from left to right, one vertex at a time. Each time a vertex is added, the cut is checked to determine whether its size is equal to one and no request in S is incomplete. By doing so, the optimal recovery point is available.

In addition to the naive approach, we use a more efficient algorithm when implementing OPTR as follows. Let wid_{inc} be the wid of the earliest incomplete write and $\mathcal{C}$ be the set of all coalescing records generated after the latest checkpoint. As in Section 3.2, a coalescing record is in the form of an $< x, y >$ pair. The pseudocode is provided in Algorithm 1.

Algorithm 1 Find Optimal Recovery Point

Input: $wid_{inc}, \mathcal{C}$
Output: the optimal recovery point

1: $wid_{rec} \leftarrow wid_{inc}$;
2: Sort $\mathcal{C}$ by x in descending order;
3: **for** $c \in \mathcal{C}$ **do**
4:     **if** $c.x < wid_{rec} \wedge c.y >= wid_{rec}$ **then**
5:         $wid_{rec} \leftarrow c.x$;
6:     **end if**
7: **end for**
8: **return** wid_{rec};

4 Filesystem and Application Optimizations

We demonstrate Ext4 filesystem and SQLite database optimizations that exploit the benefits of OPTR SSDs as follows.

At the Ext4 level, we optimize an existing filesystem primitive, *fsync* (and its variant, *fdatasync*) and introduce a new filesystem primitive, *ffence* (and its variant, *fdatafence*). For brevity, we refer to *fdatasync* as *fsync* and *fdatafence* as *ffence* in this section. Conventional *fsync* issues two flush commands to a disk, one after *fsync* transfers a journal to the disk and the other after *fsync* transfers a commit record to the disk. Our optimized *fsync* uses the same order to transfer the journal and commit record to the disk but only issues the second flush request to the disk. Conventional *fsync* requires the first flush to prevent SSDs from persisting the commit record prior to the journal. With OPTR SSDs, which preserve order, the first flush becomes safely omissible. Note that the second flush of the optimized *fsync* still can guarantee the same durability semantics as the original *fsync* does.

The newly introduced *ffence* resembles the optimized *fsync*. This primitive also uses the same order to transfer the journal and commit record to the disk but omits both flush requests. Therefore, *ffence* is a pure barrier for applications to define

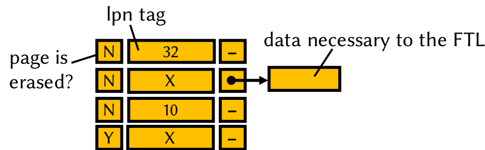


Figure 7: Emulated flash pages in VST.

the required partial order of transferring write requests to disks. With OPTR SSDs and the newly introduced *ffence*, applications can use *fsync* sparingly only when immediate durability is needed.

At the SQLite level, we modify SQLite to use *ffence* whenever a barrier alone is sufficient and to use the optimized *fsync* when immediate durability is required. To commit a single INSERT transaction, SQLite (version 3.19) calls *fsync* four times in its default configuration (journal_mode=DELETE and synchronous=FULL) [1]. Since the default configuration fails to guarantee durability on Ext4 (because unlink is not synchronous [3]), we set the configuration to the synchronous=EXTRA mode, which incurs one additional *fsync* on the parent directory after unlink to persist the deletion [4]. The objectives of the first four *fsync*s are as barriers instead of requesting immediate durability. The first *fsync* is a barrier after writing the content of a rollback journal file. The second *fsync* is a barrier to enforce that the rollback journal file exists in the directory before the db file is modified. The third *fsync* is a barrier after writing the header of the rollback journal. The fourth *fsync* is a barrier after writing the db file. Therefore, we change these four *fsync*s to *ffence*s. For the fifth *fsync*, we change it to the optimized *fsync*.

Some applications and filesystems, such as SQLite and OptFS, do not strictly demand immediate durability. For example, SQLite developers deliberately choose to avoid the fifth *fsync* by default and to allow loss of durability following a power loss event [3]. In this case, the *OPTR-enabled no-barrier mode* is the best solution.

5 OPTR Design Validation

Validating the functional correctness and crash guarantees of an FTL poses two challenges to FTL designers. First, the validation process is inherently time consuming since it requires extensive stress tests to create a large number of crash points. FTL operations such as GC, which are not invoked until a sufficient number of writes occur, make the issue even worse. Second, one may lack necessary hardware support during development of a new FTL. For example, the OpenSSD platform [2] on which we implement OPTR does not allow access to the spare area of flash, which OPTR demands during recovery.

To address this issue, we extend an FTL testing framework named Virtual Stress Testing (VST) [31] and use it to validate our OPTR implementation. Please note that we still evaluate

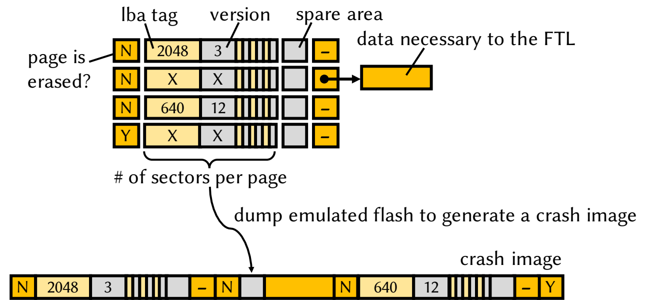


Figure 8: Extensions for crash-guarantee validation.

the performance on real OpenSSD hardware [2]. We use the same FTL code in the two experimental environments and make slight modifications to overcome hardware limitations (e.g., accessing the spare area).

5.1 VST Testing Framework

VST [31] is a simulation framework designed for validating the functional correctness of FTLs. This framework enables one to execute FTLs on PCs or servers that emulate the SRAM, DRAM and flash required by the FTLs. VST outperforms traditional FTL stress tests that use real SSDs by orders of magnitude. To validate an FTL, VST issues an extensive number of read and write requests to the FTL and reports a bug if the FTL violates any predefined rule such as incorrect page contents or a nonsequential program within a block.

The core data structure of VST is the emulated flash memory. As shown in Figure 7, an emulated flash page contains three fields: an $erased$ field that denotes whether a page has been erased, an lpn tag that records the LPN of the page, and a $data$ pointer that points to the FTL metadata stored in the page. The design of storing an lpn tag for host data rather than the actual contents dramatically decreases the memory requirement for simulation and speeds up the test. Thus, VST is particularly suitable for our validation purposes, as we want to generate a very large number of crash images.

5.2 Extending VST for Crash Recovery Tests

We largely extend VST to support crash recovery tests, which the original VST does not touch at all. For example, Figure 8 depicts the extensions we made to the emulated flash memory. We divide each emulated page into multiple 512-byte sectors to match the granularity of IO requests and further add a $version$ attribute to each emulated sector. The $version$ of a sector starts with 0 and increments by 1 when the sector is updated. For crash simulation, we fork a separate thread to periodically take snapshots of the emulated flash memory and store the snapshots as crash images.

The process of validating whether an FTL obeys prefix semantics and request atomicity is described as follows. First, we compile the FTL under test to a Linux shared object, execute the FTL on the extended VST, drive the FTL using

a trace file, and generate a number of crash images. Then, VST deserializes each crash image back into the form of the emulated flash memory and triggers the recovery procedure of the FTL. After recovery is done, VST queries the firmware for the last write it has recovered, which we call the *recovery point*, and replays the same trace separately until the recovery point to construct the *golden disk*, which represents the result of bug-free and crash-free execution. Finally, for each written sector in the golden disk, VST issues a read targeting the sector and checks if the LBA and version returned by the FTL match with those in the golden disk. If any inconsistency exists, our extended VST framework reports a violation.

We can also validate whether the FTL obeys flush semantics. We insert flushes into the request sequence and record the ID of the last flushed write in the header of a crash image. If the FTL fails to recover any write prior to the last flush, our VST framework also reports a violation.

5.3 Validation Results

We select 12 write-heavy traces from the MSRC I/O traces [35], which cover a variety of server-level access patterns [46], to drive the FTL tests. Our validation process consists of three runs. In the first run, we execute the firmware without crashes being simulated to validate its functional correctness. Similar to the validation approach in [31], we repeat each trace until the write amount reaches 1 TB. In the second run, we let the firmware recover from a total of 2,400 crash images created without any flush issued to validate whether the FTL correctly obeys prefix semantics and request atomicity. The final run is similar to the previous run, but we additionally insert a flush for every 1,000 writes to validate whether the FTL also obeys the flush semantics.

Table 1 shows the final validation results. Our implemented OPTR passes all three runs of tests. Please note that these results also suggest that our extended VST simulation framework passes the stress test, and this finding makes the simulator a much more reliable platform for future research.

Table 1: Validation results for our OPTR FTL.

	Functional Correctness	Prefix Semantics Preserved	Flush Semantics Obeyed
1-TB stress test	V		
2400 images w/o flushes		V	
2400 images w/ flushes		V	V

6 Evaluation

6.1 Experimental Setups

We implement the OPTR FTL on a real SSD (OpenSSD [2]) with an ARM7 core at 87.5 MHz, 96 KB on-chip SRAM, 64 MB DRAM, and 128 GB flash memory. We organize a total of 16 flash chips into four channels, each channel connects to four chips, each chip contains approximately 4,000 blocks, and each block contains 128 16-KB pages. The SSD adopts a static allocation strategy, which allocates a logical page to a certain flash chip based on modulo [22]. We allocate 8 MB of DRAM as the write cache of the SSD and adopt the LRU cache replacement policy. We adopt the greedy policy for GC [29, 42]. We store the per page recovery information (i.e., the wid, $size$ and lpn described in Section 3.1) in the last page of each flash block. For the OPTR-specific settings, we reserve 32 blocks for full checkpoints and 32 blocks for incremental checkpoints and coalescing records.

All the experiments are performed on a server with a 6-core Intel i7-8700 CPU and 32 GB DRAM running Ubuntu 16.04. We erase the entire SSD before conducting each experiment. More experimental setups are described in the captions of each experimental results figure.

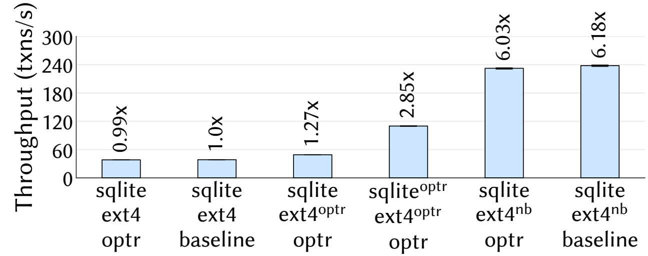


Figure 9: Performance of the three OPTR usage modes. In addition to the three usage modes and the baseline, we plot two additional bars: The leftmost bar represents running unmodified SQLite and Ext4 on OPTR to demonstrate OPTR's overhead for enforcing ordering. The rightmost bar represents running unmodified SQLite and Ext4 mounted with the no-barrier option; this configuration guarantees neither durability nor consistency and is for demonstrating the upper bound.

6.2 System-Level Performance

In this set of experiments, we evaluate three usage modes of OPTR as mentioned previously. The first mode (② in Figure 2) runs unmodified SQLite and the optimized version of Ext4; the second mode (③ in Figure 2) runs the optimized versions of SQLite and Ext4 (details of the optimizations are described in Section 4). The third mode is the OPTR-enabled no-barrier mode (④ in Figure 2). This mode guarantees consistency but achieves only eventual durability instead of immediate durability.

We use a microbenchmark that keeps generating transactions for a time interval of five minutes. Each transaction inserts a key-value pair into the SQLite database. Figure 9 shows the throughput performance of the three usage modes of OPTR. Removing the first flush of fdatasync (the third bar) improves the performance by 1.27×; invoking fdatafences (the fourth bar) for ordering constraints yields 2.85× im-

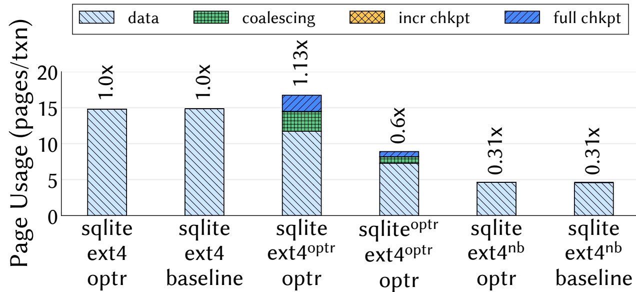


Figure 10: Endurance impact of the three OPTR usage modes.

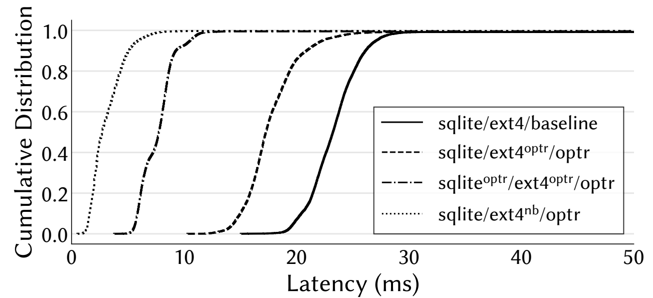


Figure 11: Latency of the three OPTR usage modes.

provement; finally, the OPTR-enabled no-barrier mode performs 6.03× better than the baseline, and this result is very close to the upper bound of 6.18×. We conduct the experiments ten times, and the errors are within ± 0.7%. To quantify the FTL overhead of enforcing the order, we compare unmodified SQLite and Ext4 atop OPTR SSDs (the first bar) with unmodified SQLite and Ext4 atop baseline SSDs (the second bar). The overhead is unnoticeable (hidden by the flushes of the workload). This result is consistent with our analysis in Section 7. However, we would like to emphasize that OPTR is not overhead-free (as shown in Section 7, OPTR can decrease the performance by up to 11.1% in some synthetic workloads).

In terms of SSD endurance impact, OPTR may improve endurance because removing flushes results in a greater chance of write coalescing; OPTR may also hurt endurance because OPTR stores checkpoints and coalescing records into flash memory for the sake of crash recovery. Figure 10 shows the overall endurance impact of the three OPTR usage modes. The y-axis denotes the average number of flash pages written per transaction. We break down written pages into user data (data) and OPTR-related data (i.e., incremental checkpoints, full checkpoints, and coalescing records).

The first usage mode of OPTR (the third bar) slightly increases the number of written pages per transaction (1.13×). The second usage mode (the fourth bar) lowers the frequency of flush operations and thereby decreases the number of written pages per transaction to 0.6×. The third usage mode does not incur any flush, and thus, the number of written pages is 0.31× that of the baseline.

Figure 11 shows the cumulative latency distribution of transactions. Flushes are slow in nature, and thus, partially or fully removing them is expected to achieve shorter latency. The experimental results show that the average latencies of OPTR in the first, second and third usage modes are 0.78×, 0.35× and 0.17× that of the baseline, respectively.

7 FTL Analyses

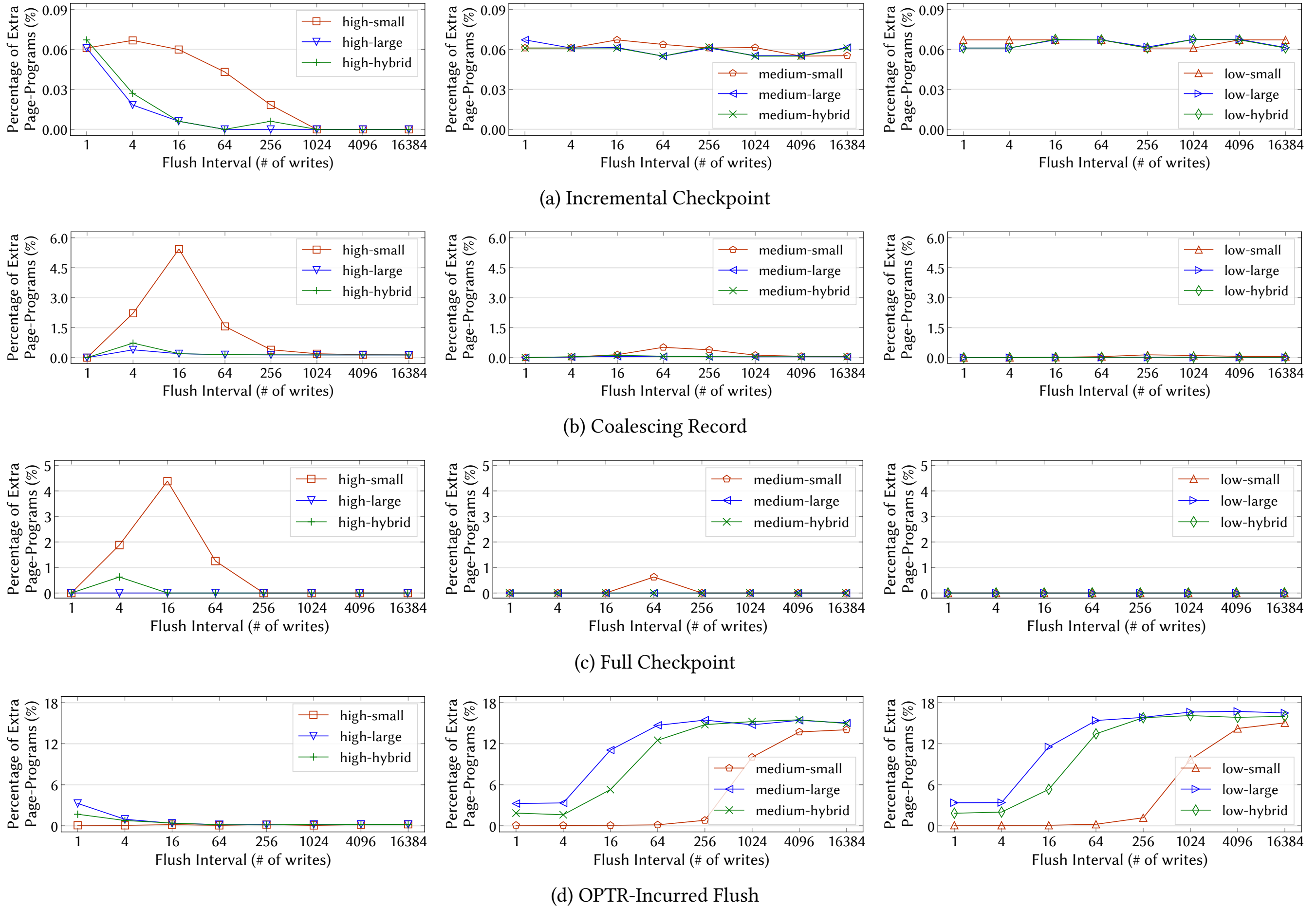
In this section, we analyze the FTL in more detail. We first report the extra flash page-programs caused by OPTR. Next, we analyze the extra GC constraints and report the overhead in terms of throughput performance. The above experiments are conducted using synthetic workloads that comprise random writes with different access localities (high: 1 MB footprint, medium: 80 MB footprint, and low: 1 GB footprint), different access granularities (large: 1 MB, small: 16 KB, and hybrid: half large and half small), and different flush intervals (1, 4, ..., to 16,384 writes). Note that these analyses are pessimistic because OPTR does not benefit from reducing flush requests in the synthetic workloads. Finally, we discuss the memory overhead (i.e., extra SRAM, DRAM, and flash space) caused by OPTR, and then, we estimate the recovery time.

7.1 Extra Flash Page-Programs

OPTR incurs extra flash page-programs because of incremental checkpoints, coalescing records, full checkpoints, and internally invoked flushes.

The overhead of **incremental checkpoints** is negligible (<0.075%), as shown in Figure 12a. The overhead is low because writing each user data page (e.g., 16 KB) incurs only an 8-byte L2P table change record, which constitutes incremental checkpoints. The overhead of incremental checkpoints is even low if write requests coalesce in the write cache. For example, the overhead is less than 0.01% for the workload with high access locality and a long flush interval (e.g., the *high-large* workload with a flush interval greater than 16 writes).

The overheads of **coalescing records** and **full checkpoints** are also negligible for most workloads. An exception is the workload with high locality and small access granularity (i.e., *high-small*) when the flush interval is between four and 64 writes (Figures 12b and 12c). High write locality tends to incur write coalescing. With more coalescing records, the checkpoint area is filled quickly, and OPTR invokes full checkpoints more frequently. If the flush interval is one write, flushes happen to suppress the occurrence of write coalescing. If the flush interval is long (e.g., >256 writes) or if the size of the write granularity is large, the overhead decreases because approximately 800 coalescing records result in an extra 16 KB page-program (each coalescing record is 20 bytes in our implementation).



(a) Incremental Checkpoint

(b) Coalescing Record

(c) Full Checkpoint

(d) OPTR-Incurred Flush

Figure 12: Page write overhead analysis using synthetic workloads. The prefix in the legends indicates the locality: *high*/*medium*/*low* access 1 MB/80 MB/1 GB of flash. The suffix indicates the access granularity: *small*/*large* write 16 KB/1 MB of data per request, and *hybrid* writes 16 KB or 1 MB of data per request, each with a 50% chance.

The percentages of page-programs caused by **internally invoked flushes** are shown in Figure 12d. In our OPTR implementation, the FTL internally flushes the write cache under the following four conditions: before an incremental checkpoint, before a full checkpoint, before GC, and every five seconds. The first three types of flushes are for correct recovery in case a crash happens, and the fourth type of flushes is for bounding the longest time duration for which dirty data can reside in the write cache. We observe a reasonable trend: if the flush interval is short (e.g., every four writes), the page-programs caused by internally invoked flushes only account for a small percentage (less than 4%). In comparison, if the flush interval is long (e.g., every 64 writes), the page-programs caused by internally invoked flushes account for a high percentage (up to 17%). Note that the page-programs analyzed here are for user data instead of OPTR's metadata, and baseline SSDs also need to flush their write caches at some points. Therefore, the above analysis is a conservative overestimation of overhead.

7.2 GC Constraints

Although OPTR constrains GC from selecting recently written flash blocks to guarantee recoverability, this constraint does not cause significantly adverse effects. The reasons are two-fold. First, OPTR constrains GC from selecting flash blocks written after the latest full or incremental checkpoint. According to our analysis, this type of blocks accounts for only 0.2% of the total blocks on average. Second, sophisticated GC policies such as age-aware GC policies avoid selecting recently written blocks in the expectation of data invalidation due to write locality.

7.3 Performance Overhead

Figures 13a, 13b, and 13c show the throughput performance (MB/s). Dotted lines denote the throughput of the baseline, and solid lines denote that of OPTR. Overall, OPTR incurs less than 2% performance overhead on average and up to 11.1% performance overhead for the synthetic workloads. Note that OPTR can slightly outperform the baseline for some workloads when OPTR's mechanisms happen to match the

workload characteristics. For example, for workloads with a high locality, age-aware GC is better than greedy GC. OPTR happens to achieve some effects of age-aware GC because OPTR constrains GC from selecting recently written flash blocks.

7.4 Memory Overhead

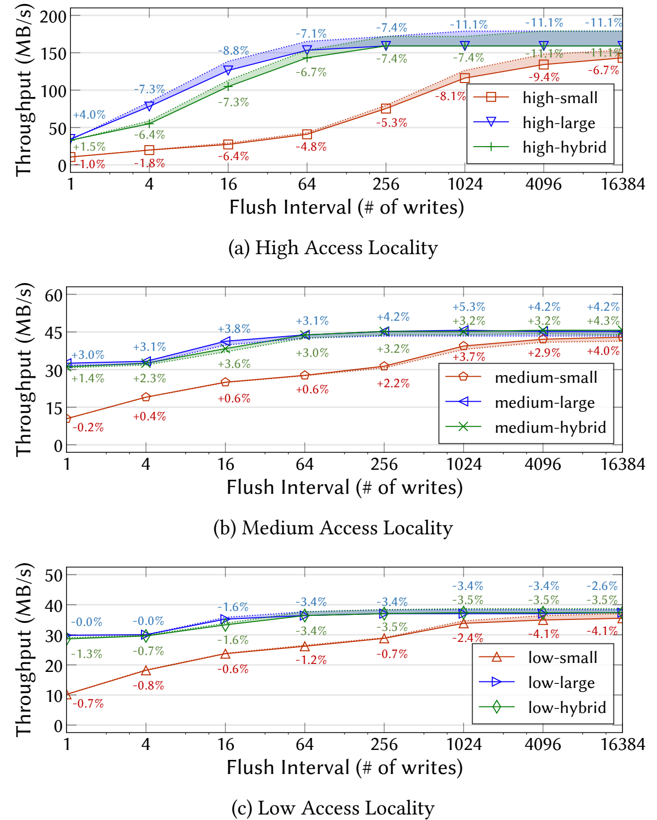
The DRAM and SRAM overheads for implementing OPTR are analyzed as follows. First, OPTR associates each entry in the write cache with two extra attributes, wid (8 bytes) and $size$ (4 bytes). In our implementation, the write cache is 8 MB (16 KB $\times$ 512 entries). Therefore, the space for storing wid is 8 $\times$ 512 = 4 KB (in DRAM in our implementation), and the space for storing $size$ is 4 $\times$ 512 = 2 KB (in SRAM in our implementation). Second, we employ a 256 KB DRAM area to buffer incremental checkpoints. Third, OPTR stores coalescing records in a DRAM buffer, whose size is equal to a flash page, e.g., 16 KB. Finally, the memory space for crash recovery can overlay that for regular FTL operation, so we do not consider this space as additional overhead.

The flash memory overhead for implementing OPTR is described as follows. First, OPTR additionally stores wid and $size$ (8 bytes and 4 bytes, respectively) in the spare area of flash memory. Modern flash memory with 16 KB page size equips each page with a 2,208-byte spare area (i.e., page size = 16,384 + 2,208 = 18,592 bytes) [6]. Since OPTR only consumes 12 bytes out of the 2,208 bytes, the overhead is only 12/2,208=0.5%, and the impact on the ratio of the error correction code rate is only 16,384/18,580 - 16,384/18,592 = 0.06%. Second, OPTR stores the lpn of each flash page in the spare area and summarizes the lpn of a block of pages in the last page of the block. We anticipate that baseline SSDs also do so. Third, OPTR keeps incremental checkpoints, coalescing records, and full checkpoints in flash memory. We set the area for storing incremental checkpoints and coalescing records to 64 MB. The size of a full checkpoint is equivalent to that of the L2P table. Given an SSD with 128 GB flash memory and 16 KB pages, the size of a full checkpoint and its shadow is at most 128 GB/16 KB $\times$ 4 B $\times$ 2 = 64 MB.

7.5 Recovery Time

OpenSSD does not provide an access approach to the spare area of flash pages, so we are not able to measure recovery time using OpenSSD. Instead, we conduct a worst-case estimation. We anticipate that accessing flash memory, which is slower than SRAM, DRAM, and computation, dominates the recovery time.

Recovery begins with reading the full checkpoint, whose size is approximately 32 MB (2,048 pages) for a 128 GB SSD. Second, OPTR sequentially scans the 64 MB area (4,096 pages) for storing incremental checkpoints and coalescing records. Third, OPTR scans the summary pages programmed after the latest checkpoint. Given a 256 KB buffer for incremental checkpoints, the number of summary pages after the latest checkpoint can be up to 256. Finally, each flash chip can



(a) High Access Locality

(b) Medium Access Locality

(c) Low Access Locality

Figure 13: Performance overhead for preserving write order.

have a partially written block, and OPTR needs to access these pages during recovery. We assume that the number of flash pages of partially written blocks is 4,096. Overall, there are 10,496 pages. We conservatively assume that reading each flash page costs 100 μs and no internal parallelism is available; then, the worse-case recovery time is 10,496 $\times$ 100 μs = 1 s, which is acceptable in general.

8 Related Work

OptFS [15], BarrierFS [45], and Featherstitch [23] are related filesystem and IO stack works that propose to decouple ordering from durability. Unlike this work, they do not focus on the FTL design and recovery process of SSDs. OptFS presents a filesystem that requires disks to support asynchronous durability notification, which notifies a host when certain blocks become persistent. BarrierFS presents a filesystem and an IO stack that require disks to support the cache barrier command, which is only available in a few eMMC (embedded multimedia card) products [45] but unavailable in the standard block device interfaces of off-the-shelf SATA, SAS, and NVMe SSDs. BarrierFS [45] implements an FTL to support the barrier write command in a commercial eMMC product. OPTR is complementary to both OptFS and BarrierFS and can simplify their designs. For instance, OptFS requires checksum encoding/decoding to enforce the ordering con-

straint between journal metadata and the commit record, and BarrierFS uses explicit cache barrier primitives to declare order; with OPTR, both can be simplified out because OPTR implicitly preserves the write order.

Mime [13], the Logical Disk [21], Stasis [43], TxFlash [40], Beyond Block IO [37], Mars [19], LightTx [32], X-FTL [28], and Isotope [44] propose transactional storage with full or partial supports to ACID at the disk level. The notion of transaction is stronger than OPTR's guarantees, but these works all propose changing the standard block device interface, which inevitably incurs significant software stack changes. Among these works, Mime [13] also advocates the benefits of request atomicity and ordering, which are in line with OPTR's design. Compared with OPTR, Mime [13] is fundamentally different because it is HDD design instead of SSD design. In addition, Mime does not allow write coalescing in a write cache, which OPTR can handle.

Increasing the disk-level crash guarantee not only can improve system performance but also, more importantly, helps to reduce crash vulnerabilities. In this sense, just replacing baseline SSDs with OPTR SSDs is beneficial. Crash vulnerabilities are serious problems. Pillai et al. [39] find 60 application-level crash vulnerabilities in widely used applications such as LevelDB and Git. Zheng et al. [47] also find ACID violations in many database systems. These vulnerabilities are mainly caused by the weak and vague crash guarantees provided by the underlying filesystems. Thus, Bornholt et al. present crash-consistency models [10], and Pillai et al. [39] specify a set of persistent properties. These approaches aim to confine and standardize the crash behaviors of filesystems, and OPTR can help to achieve these aims.

Several studies have attempted to reduce the usage or overhead of flush operations at the application or filesystem level. BarrierFS [45] and [5] replace *unnecessary flushes* with cache barrier commands. Our previous work-in-progress report [12] proposes to directly omit *unnecessary flushes* by considering *order-preserving SSDs* that achieve *strong request-level crash guarantees*. iJournaling [38] performs fine-grained journaling per file to mitigate the interference between fsync-intensive threads. NoFS [16] proposes backpointer-based consistency to fully eliminate ordering constraints but at the cost of not being able to implement atomic operations (e.g., rename). Xsyncfs [36] introduces external synchrony, in which a write is not immediately persisted (i.e., asynchronous writes) unless an external observer sees the write; thereby, this method provides the simplicity of synchronous writes and approaches the performance of asynchronous writes. Chen et al. identify the sync amplification issue in virtualized environments and propose solutions using journaling techniques at the virtual-disk level [14].

One can equip SSDs with supercapacitors, which may help to preserve write order. However, they are not widely adopted. In previous work [48], power interrupt tests are performed on 15 SSDs, and only four out of the 15 tested SSDs are equipped with supercaps. In addition, supercapacitors may be sufficient to protect an FTL from corruption, but they may not be sufficient to preserve write order. Among the four tested supercapped SSDs in [48], two still exhibit shorn or unserializable writes under power faults. Finally, capacitors are sensitive to temperature- and aging-related degradation and failures [25].

9 Conclusion

In common practice, consumer-grade SSDs (whose write cache is not battery-backed) cannot guarantee the order and atomicity of write requests upon a crash because SSD performance optimization strategies including write caching, write coalescing, request scheduling, and parallel flash programming all tend to break the guarantee. The lack of a strong crash guarantee at the disk level complicates the design of applications and filesystems and degrades the overall system performance. By exploiting the fact that SSDs adopt out-of-place updates, this work proposes order-preserving translation and recovery design (OPTR) that maintains SSD performance while preserving an illusion that write requests are completed in order and atomically. We realize the required address translation, garbage collection, and crash recovery techniques internal to a real SSD to achieve OPTR. We also develop a functional simulator to validate the correctness of OPTR.

Three usage modes of OPTR SSDs are identified and evaluated: 1) optimizing filesystems to remove the *unnecessary flushes* of *fdatasync*, 2) optimizing both applications and filesystems to further replace *fdatasync* with *fdatafence* primitives, and 3) combining the performance advantages of the no-barrier mode of filesystems and the *strong request-level crash guarantees* of OPTR SSDs. Real system experiments based on SQLite, Ext4, and a real SSD show that these three modes achieve 1.27×, 2.85×, and 6.03× performance improvement, respectively.

This work is the first SSD work with *strong request-level crash guarantees* and the standard block device interface. In comparison with previous works on transactional SSDs, we change the impression that increasing SSDs' crash guarantees is typically available at the cost of altering the standard block interface. We anticipate that OPTR can inspire more future application and filesystem designs.

Acknowledgments

We thank our shepherd, Youjip Won, and the anonymous reviewers for their valuable feedback. This research is supported in part by NOVATEK Fellowship and in part by the Ministry of Science and Technology (MOST) of Taiwan under grants 108-2218-E-007-023, 108-2218-E-007-021, and 107-2218-E-007-001. We also thank the National Center for High-performance Computing (NCHC) of Taiwan for computer time and facilities.

References

[1] Atomic commit in SQLite. https://www.sqlite.org/atomiccommit.html.

[2] OpenSSD (www.openssd-project.org/).

[3] Potential bug in crash-recovery code: unlink() and friends are not synchronous. http://sqlite.1065341.n5.nabble.com/Potential-bug-in-crash-recovery-code-unlink-and-friends-are-not-synchronous-td68885.html.

[4] Pragma statements supported by SQLite. https://www.sqlite.org/pragma.html#pragma_synchronous.

[5] Using cache barriers in lieu of REQ_FLUSH | REQ_FUA for eMMC 5.1. https://www.spinics.net/lists/linux-ext4/msg48992.html.

[6] SpecTek NAND flash detail: TLC 8192Gb. https://www.spectek.com/menus/flash_detail.aspx?memType=TLC+8192Gb, 2019.

[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC '08)*, Boston, Massachusetts, USA, 2008.

[8] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, April 2007.

[9] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, California, USA, 2017.

[10] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, Georgia, USA, 2016.

[11] Yu-Ming Chang, Ping-Hsien Lin, Ye-Jyun Lin, Tai-Chun Kuo, Yuan-Hao Chang, Yung-Chun Li, Hsiang-Pang Li, and KC Wang. An efficient sudden-power-off-recovery design with guaranteed booting time for solid state drives. In *Proceedings of the 8th IEEE International Memory Workshop (IMW '16)*, Paris, France, 2016.

[12] Yun-Sheng Chang and Ren-Shuo Liu. Improving the performance of SQLite and Ext4 using order-preserving SSDs. In *17th USENIX Conference on File and Storage Technologies Work-in-Progress Reports (FAST WiPs '19)*, Boston, Massachusetts, USA, 2019.

[13] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, Er Stepanov, John Wilkes, Richard Wagner, and Scene I. Mime: A high performance parallel storage device with strong recovery guarantees. Technical Report HPL–CSP–92–9 rev 1, Hewlett-Packard Laboratories, 1992.

[14] Qingshu Chen, Liang Liang, Yubin Xia, Haibo Chen, and Hyunsoo Kim. Mitigating sync amplification for copy-on-write virtual disk. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST '16)*, Santa Clara, California, USA, 2016.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, Pennsylvania, USA, 2013.

[16] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, California, USA, 2012.

[17] Jin-Yong Choi, Eyee Hyun Nam, Yoon Jae Seong, Jin Hyuk Yoon, Sookwan Lee, Hong Seok Kim, Jeongsu Park, Yeong-Jae Woo, Sheayun Lee, and Sang Lyul Min. Hil: A framework for compositional FTL development and provably-correct crash recovery. *ACM Trans. Storage*, 14(4):36:1–36:29, December 2018.

[18] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, 2012.

[19] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, Pennsylvania, USA, 2013.

[20] Jonathan Corbet. Barriers and journaling filesystems. https://lwn.net/Articles/283161/, 2008.

[21] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, North Carolina, USA, 1993.

[22] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and

Myoungsoo Jung. Exploiting intra-request slack to improve SSD performance. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, 2017.

[23] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, USA, 2007.

[24] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2), May 2000.

[25] Feng Gao and Dave Verburg. SSD's reliability failure mode and it's supercapacitor failure. In *Proceedings of South East Asia Technical Conference on Electronics Assembly Technologies*, Penang, Malaysia, 2016.

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13)*, San Jose, California, USA, 2013.

[27] Myoungsoo Jung, Ellis H. Wilson, III, and Mahmut Kandemir. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA '12)*, Portland, Oregon, USA, 2012.

[28] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD '13)*, New York, New York, USA, 2013.

[29] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the 1995 USENIX Technical Conference (TCON '95)*, New Orleans, Louisiana, USA, 1995.

[30] Dongwook Kim, Youjip Won, Jaehyuk Cha, Sungroh Yoon, Jongmoo Choi, and Sooyong Kang. Exploiting compression-induced internal fragmentation for power-off recovery in SSD. *IEEE Transactions on Computers*, 65(6):1720–1733, June 2016.

[31] Ren-Shuo Liu, Yun-Sheng Chang, and Chih-Wen Hung. VST: A virtual stress testing framework for discovering bugs in SSD flash-translation layers. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '17)*, Irvine, California, USA, 2017.

[32] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD '13)*, Asheville, North Carolina, USA, 2013.

[33] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[34] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Trans. Comput.*, 60(5):653–666, May 2011.

[35] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, November 2008.

[36] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3):6:1–6:26, September 2008.

[37] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA '11)*, Washington, DC, USA, 2011.

[38] Daejun Park and Dongkun Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, California, USA, 2017.

[39] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, USA, 2014.

[40] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, USA, 2008.

[41] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[43] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, USA, 2006.

[44] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: ACID transactions for block storage. *ACM Trans. Storage*, 13(1):4:1–4:25, February 2017.

[45] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, California, USA, 2018.

[46] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, California, USA, 2013.

[47] 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 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, USA, 2014.

[48] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, California, USA, 2013.

Optimizing CNN Model Inference on CPUs

Yizhi Liu[*], *Yao Wang*[*], *Ruofei Yu, Mu Li, Vin Sharma, Yida Wang*
Amazon Web Services
{yizhiliu, wayao, yuruofei, mli, vinarm, wangyida}@amazon.com

Abstract

The popularity of Convolutional Neural Network (CNN) models and the ubiquity of CPUs imply that better performance of CNN model inference on CPUs can deliver significant gain to a large number of users. To improve the performance of CNN inference on CPUs, current approaches like MXNet and Intel OpenVINO usually treat the model as a graph and use the high-performance libraries such as Intel MKL-DNN to implement the operations of the graph. While achieving reasonable performance on individual operations from the off-the-shelf libraries, this solution makes it inflexible to conduct optimizations at the graph level, as the local operation-level optimizations are predefined. Therefore, it is restrictive and misses the opportunity to optimize the end-to-end inference pipeline as a whole. This paper presents *NeoCPU*, a comprehensive approach of CNN model inference on CPUs that employs a full-stack and systematic scheme of optimizations. *NeoCPU* optimizes the operations as templates without relying on third-parties libraries, which enables further improvement of the performance via operation- and graph-level joint optimization. Experiments show that *NeoCPU* achieves up to 3.45× lower latency for CNN model inference than the current state-of-the-art implementations on various kinds of popular CPUs.

1 Introduction

The growing use of Convolutional Neural Network (CNN) models in computer vision applications makes this model architecture a natural focus for performance optimization efforts. Similarly, the widespread deployment of CPUs in servers, clients, and edge devices makes this hardware platform an attractive target. Therefore, performing CNN model inference efficiently on CPUs is of critical interest to many users.

The performance of CNN model inference on CPUs leaves significant room for improvement. Performing a CNN model inference is essentially executing a computation graph consisting of operations. In practice, people normally use high-performance kernel libraries (e.g. Intel MKL-DNN [27] and OpenBlas [51]) to obtain high performance for CNN operations. While these libraries tune very carefully for common operations with normal input data shapes (e.g. 2D convolutions), they only focus on the (mostly, convolution) operations but miss the opportunities to further optimize the end-to-end model inference at the graph level. The graph-level optimization is often handled by the deep learning frameworks, e.g. TensorFlow [5] and MXNet [8].

However, the graph-level optimization such as operation fusion and data layout planing that a framework can do is limited because the operation implementation is already predefined in the third-party libraries. Therefore, the optimizations in the frameworks do not work in concert with the optimizations in the kernel library, which leaves significant performance gains unrealized in practice. Furthermore, different CPU architectures rely on different high-performance libraries and integrating a library into a deep learning framework requires error-prone and time-consuming engineering effort. Lastly, although those libraries are highly optimized, they present as third-party plug-ins, which may introduce contention issues with other libraries in the framework. As an example, TensorFlow originally used the Eigen library [4] to handle computation on CPUs. Later on, MKL-DNN was also introduced. As a consequence, at runtime MKL-DNN threads coexist with Eigen threads, resulting in resource contention. In summary, this kind of *framework-specific* approach for CNN model inference on CPUs is inflexible, cumbersome, and sub-optimal.

Because of the constraint imposed by the framework, optimizing the performance of CNN model inference end-to-end without involving a framework (i.e. a *framework-agnostic* method) is of obvious interest to many deep learning practitioners. Recently, Intel launched a universal CNN model inference engine called OpenVINO Toolkit [16]. This toolkit optimizes CNN models in the computer vision domain on Intel processors (mostly x86 CPUs) and claims to achieve better

[*]Equal contribution

performance than the deep learning frameworks alone. Yet, OpenVINO could only provide limited graph-level optimization (e.g. operation fusion as implemented in ngraph [15]) as it still relies upon MKL-DNN to deliver performance gains for the carefully-tuned operations. Therefore, the optimization done by OpenVINO is still not sufficient for most of the CNN models.

Based on the previous observation, we argue that in order to further improve the CNN model inference performance on CPUs, being able to do the *flexible end-to-end optimization* is the key. In this paper, we propose *NeoCPU*, a comprehensive approach to optimize CNN models for efficient inference on CPUs. *NeoCPU* is full-stack and systematic, which includes operation- and graph-level joint optimizations and does not rely on any third-party high-performance libraries. At the operation level, we follow the well-studied techniques to optimize the most computationally-intensive operations like convolution (*CONV*) in a *template*, which is applicable to different workloads on multiple CPU architectures and enables us for flexible graph-level optimization. At the graph level, in addition to the common techniques such as operation fusion and inference simplification, we coordinate the individual operation optimizations by manipulating the data layout flowing through the entire model for the best end-to-end performance. In summary, *NeoCPU* does the end-to-end optimization in a flexible and automatic fashion, while the existing works rely on third-party libraries and lack comprehensive performance tuning.

NeoCPU is built upon a deep learning compiler stack named TVM [9] with a number of enhancements. TVM enables the possibility of using own operation-level optimizations instead of third-party high-performance libraries, which make it flexible to apply our operation- and graph-level joint optimization. However, there was only one customized operation-level optimization on ARM CPUs for convolutions with specific data shapes and no operation- and graph-level joint optimization in the original TVM stack before our work. In addition, there exist other deep learning compilers such as Tensor Comprehensions [46] and Glow [40]. Unfortunately, they either do not target on CPUs or not optimize the CPU performance well, e.g. based on the paper description and our own experiments, Glow only optimizes the single-core performance for CPUs. Therefore we do not incorporate those works as the baseline. Table 1 summarizes the features of *NeoCPU* compared to others. To the best of our knowledge, *NeoCPU* achieves competitive performance for CNN model inference on various kinds of popular CPUs.

Specifically, this paper makes the following contributions:

- Provides an operation- and graph-level joint optimization scheme to obtain high CNN model inference performance on different popular CPUs including Intel, AMD and ARM, which outperforms the current state-of-the-art implementations;

	Op-level opt	**Graph-level opt**	**Joint opt**	**Open-source**
NeoCPU	✓	✓	✓	✓
MXNet [8]/TensorFlow [5]	3rd party	limited	✗	✓
OpenVINO [16]	3rd party	limited	?	✗
Original TVM [9]	incomplete	✓	✗	✓
Glow [40]	single core	✓	✗	✓

Table 1: Side-by-side comparison between *NeoCPU* and existing works on CNN model inference

- Constructs a template to achieve good performance of convolutions, which is flexible to apply to various convolution workloads on multiple CPU architectures (x86 and ARM) without relying on high-performance kernel libraries;
- Designs a global scheme to look for the best layout combination in different operations of a CNN model, which minimizes the data layout transformation overhead between operations while maintaining the high performance of individual operations.

It is worth noting that, this paper primarily deals with direct convolution computation, while *NeoCPU* is compatible to other optimziation works on the computationally-intensive kernels, e.g. *CONV*s via Winograd [7, 29] or FFT [52].

We evaluated *NeoCPU* on CPUs with both x86 and ARM architectures. In general, *NeoCPU* delivers the best performance for 13 out of 15 popular networks on Intel Skylake CPUs, 14 out of 15 on AMD EYPC CPUs, and all 15 models on ARM Cortex A72 CPUs. It is worthwhile noting that the baselines on x86 CPUs were more carefully tuned by the chip vendor (Intel MKL-DNN) but the ARM CPUs were less optimized. While the selected framework-specific (MXNet and TensorFlow) and framework-agnostic (OpenVINO) solutions may perform well on one case and less favorably on the other case, *NeoCPU* runs efficiently across models on different architectures.

In addition, *NeoCPU* produces a standalone module with minimal size that does not depend on either the frameworks or the high-performance kernel libraries, which enables easy deployment to multiple platforms. *NeoCPU* is used in Amazon SageMaker Neo Service [1], enabling model developers to optimize for inference on CPU-based servers in the cloud and devices at the edge. Using this service, a number of application developers have deployed CNN models optimized for inference in production on several types of platforms. All source code has been released to the open source TVM project[2].

The rest of this paper is organized as follows: Section 2 reviews the background of modern CPUs as well as the typical CNN models; Section 3 elaborates the optimization ideas that we propose and how we implement them, followed by evaluations in Section 4. We list the related works in Section 5 and summarize the paper in Section 6.

[1] https://aws.amazon.com/sagemaker/neo/
[2] https://github.com/dmlc/tvm

2 Background

2.1 Modern CPUs

Although accelerators like GPUs and TPUs demonstrate their outstanding performance on the deep learning workloads, in practice, there is still a significant number of deep learning computation, especially model inference, taking place on the general-purpose CPUs due to the high availability. Currently, most of the CPUs equipped on PCs and servers are manufactured by Intel or AMD with x86 architecture [1], while ARM CPUs with ARM architecture occupy the majority of embedded/mobile device market [2].

Modern CPUs use thread-level parallelism via multi-core [21] to improve the overall processor performance given the diminishing increasing of transistor budgets to build larger and more complex uniprocessor. It is critical to avoid the interference among threads running on the same processor and minimize their synchronization cost in order to have good scalability on multi-core processors. Within the processor, a single physical core achieves the peak performance via the SIMD (single-instruction-multiple-data) technique. SIMD loads multiple values into wide vector registers to process together. For example, Intel introduced the 512-bit Advanced Vector Extension instruction set (AVX-512), which handles up to 16 32-bit single precision floating point numbers (totally 512 bits) per CPU cycle. And the less advanced AVX2 processes data in 256-bit registers. In addition, these instruction sets utilize the Fused-Multiply-Add (FMA) technique which executes one vectorized multiplication and then accumulates the results to another vector register in the same CPU cycle. The similar SIMD technique is embodied in ARM CPUs as NEON [3]. As shown in the experiments, our proposed solution works on both x86 and ARM architectures.

In addition, it is worth noting that modern server-side CPUs normally supports hyper-threading [37] via the simultaneous multithreading (SMT) technique, in which the system could assign two virtual cores (i.e. two threads) to one physical core, aiming at improving the system throughput. However, the performance improvement of hyper-threading is application-dependent [35]. In our case, we do not use hyper-threading since one thread has fully utilized its physical core resource and adding one more thread to the same physical core will normally decrease the performance due to the additional context switch. We also restrict our optimization within processors using the shared-memory programming model as this is the typical system setting for CNN model inference. The Non-Uniformed Memory Access (NUMA) pattern occurred in the context of multiple processors on the same motherboard is beyond the scope of this paper.

2.2 Convolutional neural networks

Convolutional neural networks (CNNs) are commonly used in computer vision workloads [23, 26, 33, 36, 41–43]. A CNN model is normally abstracted as a computation graph, essentially, Directed Acyclic Graph (DAG), in which a node represents an operation and a directed edge pointing from node X to Y represents that the output of operation X serves as (a part of) the inputs of operation Y (i.e. Y cannot be executed before X). Executing a model inference is actually to flow the input data through the graph to get the output. Doing the optimization on the graph (e.g. prune unnecessary nodes and edges, pre-compute values independent to input data) could potentially boost the model inference performance.

Most of the computation in the CNN model inference attributes to *convolutions (CONVs)*. These operations are essentially a series of multiplication and accumulation, which by design can fully utilize the parallelization, vectorization and FMA features of modern CPUs. Existing works [19, 24, 27] have demonstrated that it is possible to achieve high performance of convolution operations on CPUs by arranging the data layout and consequently, the computation, in an architecture-friendly way. The remaining challenge is how to manage the data layout flowing through these operations efficiently to get the high performance out of the end-to-end CNN model inference.

The rest of the CNN workloads are mostly memory-bound operations associated to *CONV*s (e.g. batch normalization, pooling, activation, element-wise addition, etc.). The common practice [9] is fusing them to *CONV*s so as to increase the overall arithmetic intensity of the workload and consequently boost the performance.

The computation graph of CNN model training has no essential difference with inference, just being larger (adding in backwards operations) and with some more computationally-trivial operations (e.g. loss function). Therefore, the optimization work done for CNN model inference is applicable to training as well.

3 Optimizations

This section describes our optimization ideas and implementations in detail. The solution presented in this paper is end-to-end for doing the CNN model inference. Our proposed solution is generic enough to work for a wide range of common CNN models as we will show in the evaluation. The basic idea of our approach is to view the optimization as an end-to-end problem and search for a globally best optimization. That is, we are not biased towards a local performance optimal of a single operation as many previous works. In order to achieve this, we first present how we optimized the computationally intensive convolution operations at low-level using a configurable template (Section 3.1). This makes it flexible to search for the best implementation of a specific convolution workload on a particular CPU architecture, and to optimize the entire computation graph by choosing proper data layouts between operations to eliminate unnecessary data layout transformation overhead (presented in Section 3.2

and 3.3).

We implemented the optimization based on the TVM stack [9] by adding a number of new features to the compiling pass, operation scheduling and runtime components. The original TVM stack has done a couple of generic graph-level optimizations including operation fusion, pre-computing, simplifying inference for batch-norm and dropout [9], which are also inherited to this work but will not be covered in this paper.

3.1 Operation optimization

Optimizing convolution operations is critical to the overall performance of a CNN workload as it takes the majority of computation. This is a well-studied problem but the previous works normally go deep to the assembly code level for high performance [24,27]. In this subsection, we show how to take advantage of the latest CPU features (SIMD, FMA, parallelization, etc.) to optimize a single *CONV* without going into the tedious assembly code or C++ intrinsics. By managing the implementation in high-level instead, it is then easy to extend our optimization from a single operation to the entire computation graph.

3.1.1 Single thread optimization

We started from optimizing *CONV* within one thread. *CONV* is computationally-intensive which traverses its operands multiple times for computation. Therefore, it is critical to manage the layout of the data fed to the *CONV* to reduce the memory access overhead. We first revisit the computation of *CONV* to illustrate our memory management scheme. A 2D *CONV* in CNN takes a 3D feature map (height $\times$ width $\times$ channels) and a number of 3D convolution kernels (normally smaller height and width but the same number of channels) to convolve to output another 3D tensor. The calculation is illustrated in Figure 1, which implies loops of 6 dimensions: *in_channel*, *kernel_height*, *kernel_width*, *out_channel*, *out_height* and *out_width*. Each kernel slides over the input feature map along the height and width dimensions, does element-wise product and accumulates the values to produce the corresponding element in the output feature map, which can naturally leverage FMA. The number of kernels forms *out_channel*. Note that three of the dimensions (*in_channel*, *kernel_height* and *kernel_width*) are reduction axes that cannot be embarrassingly parallelized.

We use the conventional notation *NCHW* to describe the default data layout, which means the input and output are 4-D tensors with *batch size* N, *number of channels* C, *feature map height* H, *feature map width* W, where *N* is the outermost and *W* is the innermost dimension of the data. The related layout of kernel is *KCRS*, in which *K*, *C*, *R*, *S* stand for the output channel, input channel, kernel height and kernel width.

Following the common practice [27,45], we organized the feature map layout as *NCHW[x]c* for better memory access patterns i.e. better cache locality, in which *c* is a split sub-dimension of channel *C* in super-dimension, and the number *x* indicates the split size of the sub-dimension (i.e. $\#channels = sizeof(C) \times sizeof(c)$, where $sizeof(c) = x$). The output has the same layout *NCHW[y]c* as the input, while the split factor can be different. Correspondingly, the convolution kernel is organized in *KCRS[x]c[y]k*, in which *c* with split size *x* and *k* with split size *y* are the sub-dimensions of input channel *C* and output channel *K*, respectively. It is worth noting that a significant amount of data transformation overhead needs to be paid to get the desired layout.

In addition to the dimension reordering, for better utilizing the latest vectorization instructions (e.g. AVX-512, AVX2, NEON, etc.), we split *out_width* to *ow_outer* and *ow_inner* using a factor *reg_n* and move the loop of *ow_inner* inside for register blocking. For example, on a CPU featured AVX-512, we can utilize its 32 512-bit width registers $ZMM_0 - ZMM_{31}$ [28] as follows. We maintain the loop hierarchy to use one ZMM register to store the kernel data while others storing the feature map. The kernel values stored in one *ZMM* register (up to 512 bits, a.k.a, 16 output channels in float32) are used to multiply with a number of input feature map values continuously stored in the DRAM via AVX-512F instructions [28], whose results are then accumulated to other *ZMM* registers storing the output values. Figure 1 illustrates this idea. For other vectorized instructions, the same idea applies but the split factor of *out_width* (i.e. *reg_n*) may change.

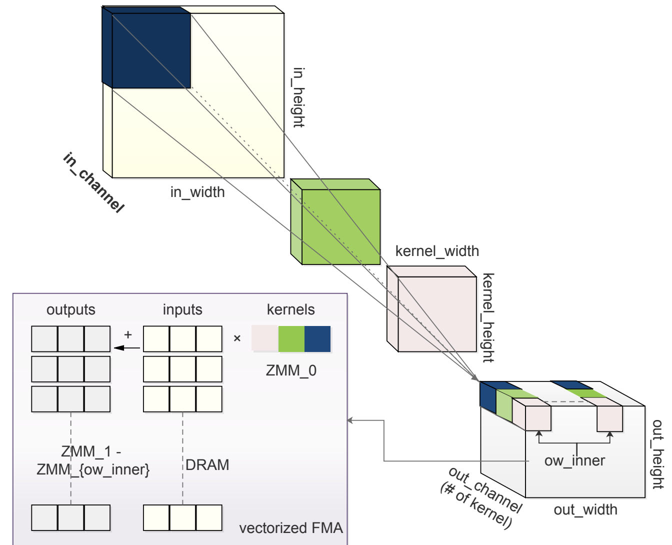


Figure 1: The illustration of *CONV* and the efficient implementation in AVX-512 instructions as an example. There are three kernels depicted in dark blue, green and light pink. To do efficient FMA, multiple kernel values are packed into one *ZMM* register and reused to multiply with different input values and accumulate to output values in different *ZMM* registers.

Algorithm 1 summarizes our optimization of *CONV* in single thread, which essentially is about 1) dimension ordering for friendly memory locality and 2) register blocking for good vectorization instruction utilization, as in previous works. However, unlike others, we made it a *template* in high-level language , in which the block size (x, y), the number of utilized registers (*reg_n*), and the loop-unroll strategy (*unroll_ker*) are easily configurable. Consequently, the computing logic can be adjusted according to different CPU architectures (cache size, registered vector width, etc.) as well as different workloads (feature map size, convolution kernel size, etc.). This is flexible and enables graph-level optimization we will discuss later.

Algorithm 1 CONV operation algorithm via FMA

```
1:  PARAM: x > 0 s.t. in_channel mod x = 0
2:  PARAM: y > 0 s.t. out_channel mod y = 0
3:  PARAM: reg_n > 0 s.t. out_width mod reg_n = 0
4:  PARAM: unroll_ker ∈ {True, False}
5:  INPUT: IFMAP in NCHW[x]c
6:  INPUT: KERNEL in KCRS[x]c[y]k
7:  OUTPUT: OFMAP in NCHW[y]c
8:  for each disjoint chunk of OFMAP do                ▷ parallel
9:      for ow.outer:= 0 → out_width/reg_n do
10:         Initialize V_REG_1 to V_REG_reg_n by 0⃗
11:         for ic.outer:= 0 → in_channel/x do
12:             for each entry of KERNEL do ▷ (opt) unroll
13:                 for ic.inner:= 0 → x do
14:                     vload(KERNEL, V_REG_0) ▷ y floats
15:                     for i:= 1 → reg_n + 1 do       ▷ unroll
16:                         vfmadd(IFMAP, V_REG_0, V_REG_i)
17:                     end for
18:                 end for
19:             end for
20:         end for
21:         for i:= 1 → reg_n + 1 do
22:             vstore(V_REG_i, OFMAP)
23:         end for
24:     end for
25: end for
```

3.1.2 Thread-level parallelization

It is a common practice to partition *CONV* into disjoint pieces to parallelize among multiple cores of a modern CPU. Kernel libraries like Intel MKL-DNN usually uses off-the-shelf multi-threading solution such as OpenMP. However, we observe that the resulting scalability of the off-the-shelf parallelization solution is not desirable (Section 4.2.4).

Therefore, we implemented a customized thread pool to efficiently process this kind of embarrassing parallelization. Basically, in a system of N physical cores, we evenly divided the outermost loop of the operation into N pieces to assign to N threads. Then we used C++11 atomics to coordinate threads during fork-join and an single-producer-single-consumer lock-free queue between the scheduler and every working thread to assign tasks. Active threads are guaranteed to run on disjoint physical cores via thread binding to minimize the hardware contention, and no hyper-threading is used as discussed in Section 2.1. For the global data structure accessed by multiple threads such as the lock-free queues, we inserted cache line padding as needed to avoid false sharing between threads. In summary, this customized thread pool employs deliberate mechanism to prevent resource contention and reduce the thread launching overhead, which makes it outperform OpenMP according to our evaluation.

3.2 Layout transformation elimination

In this subsection, we extend the optimization scope from a single operation to the entire computation graph of the CNN model. The main idea here is to come up with a generic solution at the graph level to minimize the data layout transformation introduced by the optimization in Section 3.1. Previous works [19, 24, 27] which focus on individual operation optimization normally do not consider about the data layout transformation overhead between highly optimized operations.

Since *NCHW[x]c* is efficient for *CONV*s which takes the majority of the CNN model computation, we should make sure that every *CONV* is executed in this layout. However, other operations between *CONV*s may only be compatible with the default layout, which makes each *CONV* transform the input data layout from default (*NCHW* or *NHWC*) to *NCHW[x]c* before the computation and transform it back at the end. This transformation introduces significant overhead.

Fortunately, from the perspective of the graph level, we can take the layout transformation out of *CONV* to be an independent node, and insert it only when necessary. That is, we eliminate the transformation taking place in the *CONV* operation and maintain the transformed layout flow through the graph as far as possible.

In order to determine if a data transformation is necessary, we first classify operations into three categories according to how they interact with the data layout as follows:

1. *Layout-oblivious* operations. These operations process the data without the knowledge of its layout, i.e. it can handle data in any layout. Unary operations like *ReLU*, *Softmax*, etc., fall in this category.
2. *Layout-tolerant* operations. These operations need to know the data layout for processing, but can handle a number of layout options. For example, *CONV*, in our case, can deal with *NCHW*, *NHWC* and *NCHW[x]c* layouts. Other operations like *Batch_Norm*, *Pooling*, etc., fall in this category as well.
3. *Layout-dependent* operations. These operations process the data only in one specific layout, that is, they do not

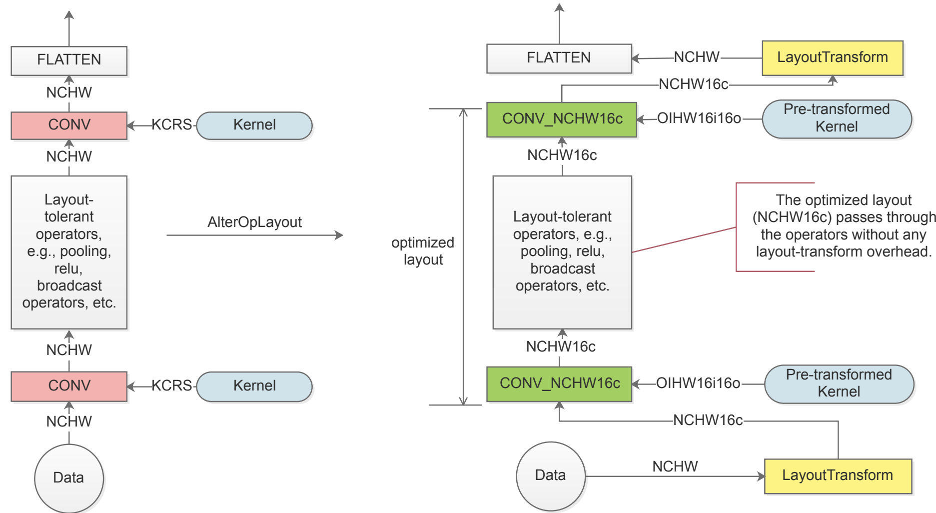


Figure 2: Layout optimization of a simple CNN model. The notation on an edge represents the layout of the data passing through this edge. The left side depicts the network with default data layout. Each *CONV* node in pink needs to pay additional overhead to transform the data into a favorable layout to achieve good performance and then transform back to default. The network in the right side is optimized at the graph level to minimize the data layout transformation during the runtime. The *CONV* nodes in green do not need to transform any data before and after computation.

tolerate any data transformation. Therefore, the layout has to be transformed to a certain format before passing to a layout-dependent operation. Transformation operations like *Flatten*, *Reshape*, etc, fall in this category.

Operations between *CONV*s in typical CNN models are either layout-oblivious (e.g. *ReLU*, *SoftMax*, *Concat*, and *ElemwiseAdd*) or layout-tolerant (e.g. *Batch_Norm*, *Pooling*), making it possible to keep the data layout being *NCHW[x]c* across convolution layers. Layout transformation from *NCHW* to *NCHW[x]c* happens before the first *CONV*. Data layout between *CONV*s can be maintained the same (i.e. *NCHW[x]c* sharing the same x value) without transformation. Only if getting to a layout-dependent operation, e.g. *Flatten*, the data layout is transformed back from *NCHW[x]c* to *NCHW*.

In practice, we first traverse the computation graph to infer the data layout of each node as illustrated in the left side of Figure 2, then we alter the layout of *CONV*s from default to *NCHW[x]c* for better performance. Note that in order to prevent further transformation, we make x a constant number (e.g. 16) across all *CONV*s. However, this value may vary across different *CONV*s in order to get the optimal performance, which requires layout transformation. We will explain more about this in Section 3.3. Finally, the *LayoutTransform* nodes are inserted to the graph accordingly. Thus, we still have *NCHW* input and output for the network, but the internal layouts between *CONV* layers are in optimized *NCHW[x]c*, as shown in the right part of Figure 2. It is worth noting that, the layout of the model parameters such as convolution kernel weights and the mean and variance of *Batch_Norm* are invariant so can be pre-transformed during the compilation. We also illustrate this in the right part of Figure 2.

We implemented the ideas by introducing multiple graph-level optimization *passes* to the TVM stack. By keeping transformed data layout invariant between *CONV* layers as much as possible and pre-transforming the layout of convolution kernel weights at compilation time, we further improve the end-to-end performance of CNN model inference.

3.3 Optimization scheme search

We came up with the aforementioned optimization schemes, especially, how to layout the data, based on our understanding of the hardware, e.g. cache size, vectorization unit width, memory access pattern, etc. However, it is tedious and impractical to exhaust all possible optimal cases by hand. As a trade-off, Section 3.2 assumes that the split factor of the channel, i.e. x in *NCHW[x]c*, stays the same during the entire network, while having various x values in different *CONV*s may lead to a better performance. In addition, the split factor of the output width, i.e. *reg_n*, also needs to adjust for different vectorization instruction sets.

Therefore, an automatic search for the best scheme is in demand to further improve the performance. Basically, we should build a system to allow the domain experts to construct the search space for the machine to explore for the best scheme resulting in the shortest execution time. The search is two-stage, first local to find optimization scheme candidates for the individual computationally-intensive operations, then global to select and combine the individual schemes for the optimal end-to-end results. It is feasible to conduct this

kind of search given the optimization template described in Section 3.1.

3.3.1 Local search

The first step is to find the optimal schedules for each computationally-intensive operations, i.e. *CONV*s in a CNN model. We used a tuple *(ic_bn, oc_bn, reg_n, unroll_ker)* to represent a convolution schedule, whose items are chosen to cover different CPU architectures and generations for different convolution workloads. The first two terms *ic_bn* and *oc_bn* stand for the split factors of input and output channels (i.e. *x* in the *NCHW[x]c* notation), which are relevant to the cache sizes of a specific CPU. The third term *reg_n* is the number of SIMD registers to be used at the inner loop, which varies among different CPU architectures and generations. Also, we observed that utilizing all SIMD registers in a single thread does not always return the best performance. The last term *unroll_ker* is a boolean deciding whether to unroll the *for* loop involving convolution kernel computation (line 12 of Algorithm 1), as in some scenarios unrolling this loop may increase the performance by reducing branch penalties and such. The local search uses the template discussed in 3.1.1 to find the best combination of these values to minimize the *CONV* execution time, similar to the kernel optimization step in [31].

Specifically, the local search works as follows:

1. Define the candidate lists of *ic_bn* and *oc_bn*. To exhaust the possible cases, we include all factors of the number of channels. For example, if the number of channels is 64, [32, 16, 8, 4, 2, 1] are listed as the candidates.
2. Define the candidate list of *reg_n*. In practice, we choose the *reg_n* value from [32, 16, 8, 4, 2].
3. Define the candidate list of *unroll_ker* to be [*True*, *False*].
4. Walk through the defined space to measure the execution time of all combinations, each of which will be run multiple times for averaging to cancel out the possible variance rooted from the unexpected interference from the operating system and/or other processes. This eventually generates a list of combinations ascendingly ordered by their execution time.

It is worth noting that we designed the above tuple in a configurable way, which means that we can always revise the tuple (e.g. adding or removing items, modifying the candidate values of an item) as needed.

Empirically, the local search of a CNN model takes a few hours using one machine, which is acceptable as it is one-time work. For example, it took about 6 hours to search for the 20 different *CONV* workloads of ResNet-50 on an 18-core Intel Skylake processor. In addition, we can maintain a database to store the results for every convolution workload (defined by the feature map and convolution kernel sizes) on every CPU type to prevent repeating search for the same convolution in different models.

Local search works well for each individual operation and indeed finds better optimization scheme than our manual work. However, greedily adopting the local optimal of every operation may not lead to the global optimal. Consider two consecutive *CONV* operations *conv_0* and *conv_1*, if the output split factor (*oc_bn*) of *conv_0* is different from the input split factor (*ic_bn*) of *conv_1*, a *LayoutTransform* node needs to be inserted to the graph as discussed in Section 3.2. This transformation overhead can be too expensive to take advantage of the benefit brought by the local optimal, especially when the data size of the network is large. On the other hand, if we maintain the same split factor throughout the entire network (as we did in Section 3.2), we may miss the opportunity to optimize some *CONV*s. Therefore, a trade-off should be made using a global search.

3.3.2 Global search

In this subsection, we extend the optimization search to the entire computation graph. The idea is to allow each *CONV* freely choosing the split factor *x* (i.e. *ic_bn* and *oc_bn*), and take the corresponding data layout transformation time into consideration. According to Section 3.2, the operations between *CONV*s are either *layout-oblivious* or *layout-tolerant*, so they can use whatever *x* decided by the *CONV* operation.

We extract a snippet of a typical CNN model in Figure 3 to illustrate the idea. From the figure we see that each *CONV* has a number of candidate schemes specified by different (*ic_bn* and *oc_bn*) pairs. The shortest execution time achieved by each pair can be obtained in the local search step. The number of pairs is bound to 100 since both *ic_bn* and *oc_bn* usually have choices less than 10. Choosing different schemes will introduce different data transformation overheads (denoted in dashed boxes between *CONV*s) or no transformation (if the *oc_bn* of the *CONV* equals the *ic_bn* of its successor). For simplicity, in the figure we omit the operations which do not impact the global search decision such as *ReLU*, *Batch_Norm* between two *CONV*s. However, operations like *Elementwise_Add* could not be omitted since it requires the layout of its two input operands (outputs of $CONV_j$ and $CONV_k$ in the figure) to be the same.

Naively speaking, if a CNN model consists of *n* *CONV*s, each of which has k_i candidate schemes, the total number of options of the global scheme will be $\prod_{i=1}^{n} k_i$, very easy to become intractable as the number of layers *n* grows. Fortunately, in practice, we can use a dynamic programming (DP) algorithm to efficiently solve this problem. Note that when choosing the scheme for a *CONV*, we only need to consider the data layout of it and its direct predecessor(s) but not any other ancestor *CONV*s as long as the so-far globally optimal schemes up to the predecessor(s) are memorized.

Therefore, a straightforward algorithm is constructed in

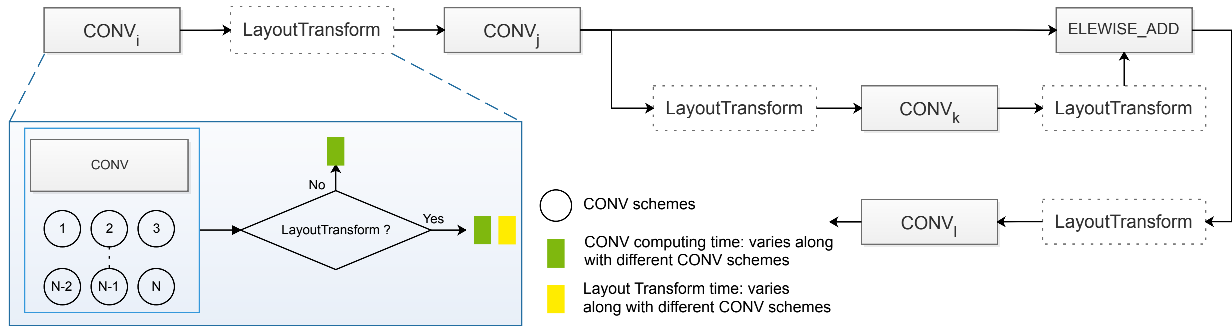


Figure 3: Global search for CNN model inference. *LayoutTransform* may or may not be in invoked according to the global decision. If invoked, an additional overhead of data transformation denoted in yellow needs to be paid.

Algorithm 2. In practice, a lot of CNN models has the structure as simple as a list, in which each *CONV* only has one predecessor [33, 41]. In this case, after a *CONV* is done, the intermediate states stored for its predecessor can be safely removed. For networks with more complex structure like using *Elementwise_Add* to add two *CONV* outputs to feed to the next *CONV* [23], it is trickier since the schemes of a *CONV* may need to be saved for a future use (e.g. in Figure 3 $CONV_l$ needs the schemes of $CONV_j$ via *Elementwise_Add*).

Algorithm 2 Global search algorithm

1: Sort the nodes of the graph in topological order
2: Initialize the optimal schemes of the *CONV*s without dependency using the execution time of their candidate schemes
3: **for** $CONV_i$ in topological order **do**
4:     **for** each candidate scheme CSI_j of $CONV_i$ **do** ▷ j is the jth scheme of $CONV_i$
5:         $t = execution_time(CSI_j)$
6:         $GSI_j = MAX$ ▷ initialize global optimal scheme of $CONV_i$ under scheme j
7:         **for** each so-far globally optimal scheme GSX_k of predecessor x **do** ▷ k is the kth scheme of $CONV_x$
8:             $cur_opt = t + transform_time(k, j) + GSX_k$
9:             **if** $cur_opt < GSI_j$ **then**
10:                 $GSI_j = cur_opt$
11:             **end if**
12:         **end for**
13:     **end for**
14: **end for**
15: **return** last node's shortest scheme

However, if the model structure becomes too complicated with many data dependency links between *CONV*s, the straightforward DP algorithm could go intractable, too. For example, in the object detection model SSD [36], the number of states can reach the order of trillions due to the occurrence of many concatenation blocks. In this case, we introduced an approximate solution to accelerate the search. Particularly, we reduced our global search problem to the register allocation problem in the canonical compiler domain with minor modification as follows. The register allocation problem is modeled as graph representation in which each node (variable) has a candidate list containing all possible register options, and each edge is associated with a cost matrix indicating the availability of registers between two nodes [20]. Similarly in our global search, each *CONV* has a list of candidate schemes and each edge is associated with the layout transformation cost matrix generated by the scheme lists of two *CONV*s. For other non-*CONV* nodes like *Elementwise_Add* which require all inputs in the same layout, we fixed the layout of one input and convert all other input layouts to it. Therefore, we defined the candidate list of a non-*CONV* node to be the same as the first input *CONV* and the cost matrix on the edge between these two nodes as all diagonal elements being 0 and all the other elements being infinite. For the edges between this non-*CONV* node and other input nodes, cost matrices are generated from the first input node and other input nodes. After such modification, all nodes and edges in our graph have the valid properties which are required by the register allocation modeling. This enables us to apply a heuristic solver based on partitioned boolean quadratic programming (PBQP) to our problem as it is done in register allocation [20].

In order to verify the result of this approximation algorithm, we compared it with the result of DP (the guaranteed best) on some simple networks where DP is tractable. It turns out that the approximation algorithm gets at least 88% of the best available result. Empirically, a typical DP search completes in 1 minute for most CNN models. In practice, we switch to the approximation algorithm if DP does not complete in 5 minutes. The approximation algorithm completes quickly, e.g. in 10 seconds. For the 15 popular networks we evaluated in Section 4, only SSD was done approximately.

4 Evaluation

This section evaluates the performance of our proposed solution, *NeoCPU*, by answering the following questions:

1. What is the overall performance of *NeoCPU* comparing

with the start-of-the-art alternatives on various kinds of CPUs?

2. What is the individual contribution of each optimization idea we proposed?

All experiments were done on Amazon EC2 instances. We evaluated *NeoCPU* on three kinds of CPUs, Intel Skylake (C5.9xlarge, 18 physical cores, featured with AVX-512), AMD EPYC (M5a.12xlarge, 24 physical cores, featured with AVX2) and ARM Cortex A72 (A1.4xlarge, 16 physical cores, featured with NEON). Although testing on the cloud, our results of ARM CPUs apply to the ones at the edge devices such as Raspberry Pi and Amazon Echo Dot due to the same architecture. All cores have uniformed memory access.

NeoCPU was built on top of the code base of the TVM stack 0.4.0. For CPUs with x86 architecture, we chose two framework-specific solutions and one framework-agnostic solution as baselines for comparison. For the framework-specific solution, we investigated a wide range of options and figured out that MXNet 1.3.1 with Intel MKL-DNN v0.15 enabled has the widest model coverage with the best inference performance compared to others (e.g. Intel Caffe). In addition, we chose TensorFlow 1.12.0 with ngraph v0.12.0-rc0 integration (empirically proved to be better than TensorFlow XLA on CPUs) due to its popularity. TensorFlow is known to have better performance on CPUs than another popular deep learning framework PyTorch [14]. The latest Intel OpenVINO Toolkit 2018 R5.445 served as the framework-agnostic solution. We used the official image-classification sample [3] and object-detection-ssd sample [4] for benchmarking. For ARM CPUs, we chose MXNet 1.3.1 with OpenBlas 0.2.18 and TensorFlow 1.12.0 with Eigen fd68453 [5] as the baselines. No framework-agnostic comparison was performed as on ARM CPUs there is no counterpart of OpenVINO to x86 CPUs. In addition, OpenMP 4.5 implemented in GCC 7.3 was used in the comparison with our own thread pool for multi-thread scalability. As a note, all implementations used direct convolution. Incorporating the advanced convolution algorithms to further improve the performance remains for future work.

We ran the model inference on a number of popular CNN models, including ResNet [23], VGG [41], DenseNet [26], Inception-v3 [43], and SSD [36] using ResNet-50 as the base network. Models consumed by MXNet and OpenVINO were from the Gluon Model Zoo [6]. Models consumed by TensorFlow were obtained mostly from TF-SLim [7] and for some missing ones (e.g. ResNet-34, DenseNet-169) we manually created them. The same model in different formats are semantically identical. As inherited from the TVM stack, *NeoCPU* is compatible to both Gluon and TF-slim formats, and in the evaluation we used the former one. The input data of the model inference are 224×224 images, except for the Inception Net (299×299) and SSD (512×512) by following the popular convention. Since the most important performance criterion of model inference is the *latency*, we did all experiments with batch size 1, i.e. each time only one image was fed to the model, to measure the inference time. Therefore, we fix the value N in $NCHW[x]c$ as 1. *NeoCPU* works for larger batch sizes as well, in which cases we just need to add the N value to our configuration tuple.

Since our optimization does not change the semantics of the model, we do not expect any change of the model output. As a sanity check, we compared the results generated by *NeoCPU* with other baselines (prediction accuracy for image classification models and mean accuracy prediction for object detection models) to validate the correctness.

4.1 Overall Performance

We first report the overall performance we got for 15 popular CNN models comparing with the baselines on different CPUs in Table 2. The results were obtained by averaging the execution times of 1000 samples, doing inference for one at a time. In general, *NeoCPU* is more efficient across different models on different CPU architectures than any of the baselines (up to 11× speedup without considering the suspicious OpenVINO outliers which will be explained later). Compared to the *best available* baseline result for each model, *NeoCPU* gets 0.94-1.15× performance on the Intel Skylake CPU, 0.92-1.72× performance on the AMD EYPC CPU, and 2.05-3.45× performance on the ARM Cortex A72 CPU.

As framework-specific solutions, MXNet and TensorFlow were suboptimal for CNN inference on CPUs because it is lacking of flexibility to perform sufficient graph level optimization (e.g. flexible data layout management). MXNet has active MKL-DNN support from Intel so it performed quite well on CPUs with the x86 architecture. MXNet performed worse than TensorFlow on ARM due to the scalability issue (demonstrated in Figure 4c). TensorFlow performs significantly worse on SSD as it introduces branches to this model, which requires dynamic decisions to be made during the runtime. Comparatively, the framework-agnostic solution provided by the OpenVINO tries to further boost the performance by removing the framework limitation. However, the performance of OpenVINO was unstable across models. Although it gets appealing results on some cases, OpenVINO sometimes performed extremely slowly on certain models (e.g. 45× slower than us for ResNet-152 on AMD) for unknown reasons. When summarizing the speedup results, we do not include these outliers. It is also worth noting that the

[3] https://docs.openvinotoolkit.org/latest/_inference_engine_samples_classification_sample_README.html
[4] https://docs.openvinotoolkit.org/latest/_inference_engine_samples_object_detection_sample_ssd_README.html
[5] https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/workspace.bzl#L128
[6] https://mxnet.incubator.apache.org/api/python/gluon/model_zoo.html
[7] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	2.77, .01	**4.85, .02**	6.60, .00	12.90, .04	18.58, .07	12.05, .00	15.16, .00	18.55, .00
TensorFlow	4.07, .00	6.95, .00	11.93, .01	20.36, .00	37.33, .02	18.78, .01	24.28, .00	27.64, .02
OpenVINO	3.54, .00	5.43, .00	7.95, .00	12.55, .00	17.32, .01	138.07, .12	137.51, .14	140.95, .33
NeoCPU	**2.64, .00**	5.14, .00	**5.73, .00**	**11.15, .01**	**17.24, .01**	**11.91, .00**	**14.91, .00**	**18.21, .00**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	21.83, .00	14.72, .00	31.07, .01	19.73, .00	26.66, .00	**10.43, .00**	42.71, .00	
TensorFlow	35.94, .00	18.65, .01	32.97, .00	23.03, .01	29.19, .01	16.39, .04	358.98, .13	
OpenVINO	147.41, .12	9.03, .00	18.55, .01	11.80, .01	14.92, .01	10.65, .00	30.25*, .01	
NeoCPU	**21.77, .00**	**8.04, .01**	**17.45, .04**	**11.21, .01**	**13.97, .03**	10.67, .01	**31.48, .00**	

(a) Overall performance on a system with 18-core Intel Skylake CPU

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	7.84, .36	14.66, .14	22.48, .48	40.57, 2.54	58.92, 3.21	49.17, 1.75	59.19, 1.35	72.57, 2.74
TensorFlow	13.95, .24	25.02, .49	38.14, .35	74.41, .56	108.38, .24	60.30, .22	71.16, .33	96.33, .22
OpenVINO	8.56, 1.02	15.18, .60	21.95, .42	1711.42, 1.59	2515.08, 2.51	662.09, 1.73	709.58, 1.78	828.17, 2.09
NeoCPU	**7.15, .49**	**14.10, .68**	**18.79, 1.01**	**39.32, .87**	**55.71, .54**	**28.58, .74**	**38.17, .29**	**57.63, .68**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	84.76, 1.91	35.00, 1.06	79.58, .63	47.82, 1.67	63.67, .15	30.12, .09	132.73, 2.59	
TensorFlow	121.04, .38	45.87, .15	98.39, .93	57.49, .28	77.37, .24	48.78, .45	747.78, 2.24	
OpenVINO	1113.17, 2.39	**22.36, .24**	818.86, 1.39	438.72, 1.27	453.12, 1.75	**25.75, .83**	93.65*, .81	
NeoCPU	**63.78, .18**	24.30, .54	**49.37, .09**	**31.70, .47**	**46.12, .51**	26.37, .32	**97.26, .54**	

(b) Overall performance on a system with 24-core AMD EYPC CPU

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	75.82, 1.31	135.24, 2.49	149.65, 2.37	252.76, 3.25	351.60, 3.49	385.50, 2.39	505.06, 3.28	575.80, 2.98
TensorFlow	50.50, .07	96.50, .11	107.50, .12	223.83, .17	336.56, .19	245.97, .18	336.05, .27	381.46, .21
NeoCPU	**19.26, .08**	**37.20, .14**	**45.73, .02**	**86.77, .08**	**126.65, .13**	**87.66, .21**	**124.75, .05**	**162.49, .14**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	642.27, 4.30	211.54, 3.22	389.33, 2.98	264.36, 3.82	315.10, 3.49	275.28, 3.27	657.22, 3.29	
TensorFlow	459.91, .27	122.48, .07	301.51, .11	159.39, .08	204.79, .10	142.00, .07	1020.16, .47	
NeoCPU	**201.03, .49**	**44.00, .09**	**87.36, .15**	**58.93, .65**	**65.48, .54**	**84.00, .08**	**318.48, .11**	

(c) Overall performance on a system with 16-core ARM Cortex A72 CPU

Table 2: Overall performance of *NeoCPU* and the selected baselines. Each entry contains the mean value of 1000 runs and the corresponding standard error. The best performance of each model is in **bold**. (*OpenVINO on Intel and AMD CPUs does not measure the entire SSD execution time)

OpenVINO measures the execution time of SSD without taking into account a significant amount of operations including *multibox detection*. Since OpenVINO is not open-sourced, we were not able to modify it for apples-to-apples comparison on the SSD model. OpenVINO does not work for ARM CPUs as it relies on MKL-DNN which optimizes only for CPUs with x86 architecture. *NeoCPU* outperforms the baselines mostly because of the advanced optimization techniques we presented in Section 3. In addition, all baselines largely rely on the third-party libraries (MKL-DNN, OpenBlas, Eigen) to achieve good performance. *NeoCPU*, on the other hand, is independent from those high-performance libraries, which gives us more room to optimize the model inference as a whole.

4.2 Optimization Implications

This subsection breaks up the end-to-end performance gain of *NeoCPU* by investigating the performance boost of each individual optimization technique we described in Section 3. For the sake of space, in each comparison we only pick one network from a network family, respectively. Other networks in the same family share the similar benefits. We only report the performance results on Intel CPUs in Section 4.2.1-4.2.3. The optimization effect applies to AMD and ARM CPUs, too. Basically, Section 4.2.1 is the operation-level optimization, and Section 4.2.2 and 4.2.3 cover the operation- and graph-level joint optimization.

4.2.1 Layout optimization of *CONV*

Firstly, we compare the performance with and without organizing the data in a memory access and vectorized instruction utilization friendly layout (*NCHW{x}c*) for the *CONV* operations at the second row of Table 3. This is the operation-level optimization that is commonly applied by the compared baselines in Section 4.1. We replicate it as a template using TVM scheduling schemes without touching the assembly code or

Speedup	ResNet-50	VGG-19	DenseNet-201	Inception-v3	SSD-ResNet-50
Baseline	1	1	1	1	1
Layout Opt.	5.34	8.33	4.08	7.41	6.34
Transform Elim.	8.22	9.33	5.51	9.11	9.32
Global Search	12.25	10.54	6.89	11.85	12.49

Table 3: The individual speedup brought by our optimization compared to the *NCHW* baseline. The speedup of row n was achieved by applying the optimization techniques till this row.

intrinsics, which enables the subsequent optimization for various CNN models on different CPU architectures. From row 2 of Table 3 we see significant improvement compared to the default data layout (*NCHW*), whose performance is normalized to baseline 1. Both implementations are with proper vectorization and thread-level parallelization, as well as basic graph-level optimizations introduced by the original TVM stack, e.g. operation fusion, pre-computing, inference simplification, etc.

4.2.2 Layout transformation elimination

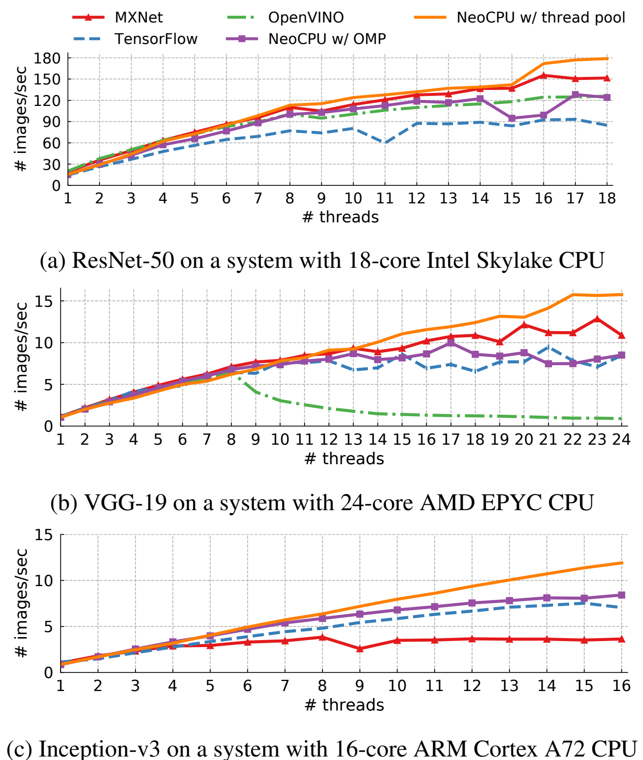
Secondly, we evaluate the performance boost brought by eliminating the data layout transformation overhead as discussed in Section 3.2. The results were summarized at the third row of Table 3. Compared to the layout optimization of *CONV* (second row of Table 3), layout transformation elimination further accelerates the execution time by $1.1-1.5\times$. *NeoCPU* uses a systematic way to eliminate the unnecessary data layout transformation by inferring the data layout throughout the computation graph and inserting the layout transformation nodes only if needed, which is not seen in other works.

4.2.3 Optimization scheme search

Next, we compare the performance between the optimization schemes produced by our search algorithm and the ones carefully picked by us manually. By comparing the third and fourth row of Table 3, our algorithm (described in Section 3.3) is able to find the (approximately) best combination of data layouts which outperforms the manually picked results by $1.1-1.5\times$. ResNet-50 (and its variants) gains more speedup from global search because the network structure is more complicated, hence leaving more optimization room. In contrast, VGG-19 (and its variants) gains less since the structure of this model is relatively simple. SSD utilizes the approximation algorithm and gets significant speedup, too. The results also verify that, with automatic search, we can get rid of the tedious manual picking of parameters by producing even better results. To the best of our knowledge, *NeoCPU* is the only one that does this level of optimization.

4.2.4 Multi-thread parallelization

Lastly, we did a strong scalability experiment using the multi-threading implementations backed by our own thread pool described at Section 3.1.2 and the commonly used OpenMP API implemented in the GCC compiler. We also included the result of MXNet, TensorFlow and OpenVINO using Intel MKL-DNN, OpenBlas or Eigen (all realizing multi-threading via OpenMP) for comparison. We configured OpenMP via environment variables to make sure that the jobs are statically partitioned and each thread runs on a disjoint core, which resemble the behavior of our thread pool for apples-to-apples comparison. Figure 4 summarizes the number of images a model can inference one by one (i.e. *batch size* $=1$) in a second as a function of the number of threads the model inference uses. For the sake of space, we demonstrate one result for one CPU type. The figure shows that our thread pool achieves better scalability than OpenMP in *NeoCPU* as well as in the baselines. Although the tasks are embarrassingly parallelizable, each model inference consists of a number of parallelization regions. The overhead of OpenMP to launch and suppress threads before and after a region is larger than our thread pool, which attributes to the less scalability of OpenMP. Furthermore, sometimes we observed that the performance obtained by OpenMP jitters, or even drops, while adding threads. In addition, the performance of OpenMP may differ across different implementations. In summary, our evaluation suggests that in our use cases, it is preferable to have a self-customized thread pool with full control.



(a) ResNet-50 on a system with 18-core Intel Skylake CPU

(b) VGG-19 on a system with 24-core AMD EPYC CPU

(c) Inception-v3 on a system with 16-core ARM Cortex A72 CPU

Figure 4: Scalability comparison between different multi-threading implementations. The standard errors (<0.4) are too small to be visible in the diagrams.

5 Related Works

As deep learning demonstrates more and more power in the real-world applications, there is a significant amount of effort being made to accelerate the deep learning workloads on all kinds of hardware ranging from CPUs [24, 27, 44, 53], GPUs [11, 13], FPGAs [18, 22, 49], to special-purpose accelerators [12, 32]. Modern deep learning frameworks normally leverage these optimized implementations to run deep learning training and inference on the corresponding hardware targets. There are also works tailored for inference to address the inference-specific requirement such as low latency and small binary size on different hardware targets (e.g. GPUs [38], ASICs [22]). *NeoCPU* is more flexible and combines the operation- and graph-level optimization intelligently. Although this paper focuses on CPUs, the ideas are applicable to other hardware targets.

NeoCPU is based on the TVM stack [9], an end-to-end framework inspired by Halide [39], which expresses a deep learning model into intermediate representations (IRs) and compiles to the machine code. There are several other similar deep learning compilers such as TensorFlow XLA [34], Tensor Comprehensions [46], Glow [40] and DLVM [47]. However, so far none of them has reported CPU inference results on par with what we did (e.g. Glow only optimized single-core performance on CPUs). We believe our proposed solution could be an integral part to these frameworks.

We follow the well-studied ideas implemented in other high-performance libraries [27, 51] to optimize the computationally-intensive *CONV* operations. In addition to the libraries, there are also highly customized optimization works for convolutions and matrix multiplications on Intel CPUs [19, 24]. These works are mostly about individual operation-level optimizations, which do not consider maintaining data layouts through the entire network. Specifically, they carefully investigate the computation nature of convolutions as well as the available CPU resources to fine tune the operations. This kind of optimization is able to maximize the convolution performance on the targeted CPUs but is not very flexible to extend to other platforms and to do joint optimization. Unlike others, we make the optimization as a configurable template so that it is flexible to fit to different CPU architectures and enable the opportunity to surpass manually tuned performance via operation- and graph-level joint optimization.

Our work utilizes auto search to look for optimal solutions. Similar *auto-tuning* ideas were used in other works as well [10, 46, 48]. However, they all focused on performance tuning for single operations, while ours extends the scope to the entire CNN model to search for optimal solutions globally. Recently, we also observed other work optimizing the DNN workloads at the graph level [30]. This work attempts to obtain better global performance using relaxed graph substitutions which may harm the local performance within a few operations. Its non-greedy search idea is conceptually similar to ours and potentially applicable to our solution. The approximation algorithm we employed to deal with the global search for the models with complicated structures (e.g. SSD) is inspired by the application of PBQP in the register allocation problem [6, 17, 20]. This paper leverages the previous idea and applies to a new domain by minor modification.

6 Conclusion

In this paper, we proposed an end-to-end solution to compile and optimize convolutional neural networks for efficient model inference on modern CPUs. The experiments show that we are able to achieve up to 3.45× speedup on 15 popular CNN models on the various kinds of CPUs (Intel Skylake, AMD EPYC and ARM Cortex A72) compared to the performance of the state-of-the-art solutions. The future work includes extending to other convolution computation algorithms such as Winograd and FFT, handling model inference in quantized values (e.g. INT8) and extending our operation- and graph-level joint optimization ideas to work on other hardware platforms (e.g. NVidia GPUs compared with TensorRT). Supporting the optimized model inference in dynamic shapes (e.g. RNNs [25, 50]) is another interesting direction to explore.

Acknowledgments

We would like to thank our shepherd Peter Pietzuch and the anonymous reviewers of the USENIX ATC program committee for their valuable comments which improved the paper a lot. We are also grateful to Tianqi Chen and Animesh Jain for helpful discussion and constructive suggestion.

References

[1] Amd vs intel market share. https://www.cpubenchmark.net/market_share.html. [Online; accessed 13-May-2019].

[2] Arm holdings. https://en.wikipedia.org/wiki/Arm_Holdings. [Online; accessed 13-May-2019].

[3] Neon. https://developer.arm.com/technologies/neon. [Online; accessed 13-May-2019].

[4] Eigen: a C++ Linear Algebra Library. http://eigen.tuxfamily.org/, 2017. [Online; accessed 13-May-2019].

[5] 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 *OSDI*, volume 16, pages 265–283, 2016.

[6] Cooper K.D. Torczon L. Briggs, P. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16(3) 428–455*, 1994.

[7] David Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. Deep tensor convolution on multicores. *arXiv preprint arXiv:1611.06565*, 2016.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018.

[11] Xie Chen, Yongqiang Wang, Xunying Liu, Mark JF Gales, and Philip C Woodland. Efficient gpu-based training of recurrent neural network language models using spliced sentence bunch. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[12] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759*, 2014.

[14] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *NIPS ML Systems Workshop*, 2017.

[15] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Krovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[16] Deanne Deuermeyer and Andrey Z. Openvino toolkit release notes. https://software.intel.com/en-us/articles/OpenVINO-RelNotes. [Online; accessed 13-May-2019].

[17] Erik Eckstein. *Code optimizations for digital signal processors*. PhD thesis, Vienna University of Technology, 2003.

[18] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. CNP: An FPGA-based Processor for Convolutional Networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009.

[19] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2018.

[20] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with pbqp. *JMLC 2006. LNCS, vol.4228, pp. 346-361*, 2016.

[21] Lance Hammond, Benedict A Hubbert, Michael Siu, Manohar K Prabhu, Michael Chen, and K Olukolun. The stanford hydra cmp. *IEEE micro*, 20(2):71–84, 2000.

[22] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *FPGA*, pages 75–84, 2017.

[23] 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.

[24] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991. IEEE, 2016.

[25] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 41. ACM, 2019.

[26] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[27] Intel. Intel math kernel library for deep neural networks (intel mkl-dnn). https://github.com/intel/mkl-dnn, 2018. [Online; accessed 13-May-2019].

[28] James R. (Intel). Intel avx-512 instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions, 2013. [Online; accessed 13-May-2019].

[29] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. Optimizing n-dimensional, winograd-based convolution for manycore cpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 109–123. ACM, 2018.

[30] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *SysML*, 2019.

[31] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. In *SysML*, 2018.

[32] 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 Gottipati, 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. ACM, 2017.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[34] Chris Leary and Todd Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[35] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 45, 2002.

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[37] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. Hyper-threading technology in the netburst® microarchitecture. *14th Hot Chips*, 2002.

[38] NVIDIA. Nvidia tensorrt. https://developer.nvidia.com/tensorrt, 2018. [Online; accessed 13-May-2019].

[39] 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. ACM, 2013.

[40] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[44] Linpeng Tang, Yida Wang, Theodore Willke, and Kai Li. Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs. *ArXiv e-prints*, July 2018.

[45] Tensorflow. Tensorflow performance guide. https://www.tensorflow.org/performance/performance_guide#data_formats, 2018. [Online; accessed 13-May-2019].

[46] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[47] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler framework for neural network dsls. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2017.

[48] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 38–38. IEEE, 1998.

[49] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Metworks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[50] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 951–965, 2018.

[51] Xianyi Zhang, Qian Wang, and Zaheer Chothia. Openblas. http://xianyi.github.io/OpenBLAS, 2014. [Online; accessed 13-May-2019].

[52] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. Fft convolutions are faster than winograd on modern cpus, here is why. *arXiv preprint arXiv:1809.07851*, 2018.

[53] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 801–811. IEEE, 2016.

Accelerating Rule-matching Systems with Learned Rankers

Zhao Lucis Li[⋆‡] Chieh-Jan Mike Liang[‡] Wei Bai[‡] Qiming Zheng[†‡]
Yongqiang Xiong[‡] Guangzhong Sun[⋆]
[⋆]*University of Science and Technology of China* [‡]*Microsoft Research* [†]*Shanghai Jiao Tong University*

Abstract

Infusing machine learning (ML) and deep learning (DL) into modern systems has driven a paradigm shift towards learning-augmented system design. This paper proposes the learned ranker as a system building block, and demonstrates its potential by using rule-matching systems as a concrete scenario. Specifically, checking rules can be time-consuming, especially complex regular expression (regex) conditions. The learned ranker prioritizes rules based on their likelihood of matching a given input. If the matching rule is successfully prioritized as a top candidate, the system effectively achieves early termination. We integrated the learned rule ranker as a component of popular regex matching engines: PCRE, PCRE-JIT, and RE2. Empirical results show that the rule ranker achieves a top-5 classification accuracy at least 96.16%, and reduces the rule-matching system latency by up to 78.81% on a 8-core CPU.

1 Introduction

Machine learning (ML) and deep learning (DL) bring new possibilities to modern system designs [9, 15, 17, 23, 24], which traditionally rely on human-written heuristics. Under the *learning-augmented design*, system logic is implemented with both heuristics and ML/DL to better address performance bottlenecks. Such design has the following advantages. First, compared to heuristics, ML/DL has been shown to excel in learning complex data patterns to enable classification, regression and prediction. Second, while traditional heuristics are designed to be general purpose and computation-efficient, systems such as web services can have a workload that is highly dynamic and scenario-specific. Adapting to workload characteristics can enable highly optimized algorithmic operations and system components.

This work was done when Zhao Lucis Li and Qiming Zheng were interns at Microsoft Research. Chieh-Jan Mike Liang is the corresponding author.

However, formulating ML/DL tasks into system building blocks is non-trivial. Given that ML/DL is stochastic in nature, inference uncertainties should not impact the system correctness. And, inference should not impose a significant resource overhead on the end-to-end system performance. Recently, the industry has had success in using learning-driven space exploration as a system building block, for scenarios such as system configuration tuning [9, 17, 23]. Building on this success, this paper explores the potential and feasibility of another building block for learning-augmented systems – the *learned ranker*.

Particularly, we use rule-matching systems as a concrete scenario for learned rankers. One common task of rule-matching systems is to match the given input to *one* rule in the ruleset as fast as possible, and the performance bottlenecks come from two observations. For rulesets that do not impose a mandatory ordering on rule checking, the naïve practice of sequentially going through rules can result in processing many unnecessary rules. The problem exacerbates when we consider that rules can have non-trivial conditions written in regular expressions (regex). Since regex matching engines typically rely on either deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA), its overhead largely depends on the length and complexity of regex patterns and inputs. In the worst case, the backtracking problem can result in $O(2^n)$ time complexity for an input string of size n [13], rather than the expected $O(n)$.

To reduce the rule matching latency, common optimization techniques include string-matching pre-filters [7, 12], just-in-time compilation [4], and specialized hardware-based regex acceleration [20, 25]. The learned ranker enables a different but complementary technique – after the string-matching pre-filter removes unlikely inputs, it performs per-input rule prioritization for the regex matching engine, based on the likelihood of a given input to match each rule. Conceptually, if the matching rule can be prioritized as one of the top candidates, the rule-matching system effectively achieves early-termination, thus minimizing unnecessary rule checking.

Designing learning-augmented systems with the learned

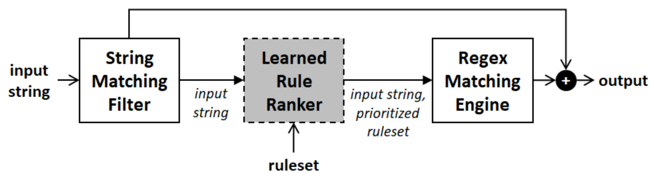


Figure 1: A learning-augmented design of rule-matching systems. Complementing existing acceleration techniques, we introduce a learned rule ranker to dynamically prioritize rules for each input. The goal is to minimize unnecessary rule processing and achieve early-termination.

ranker as a building block should go beyond simply selecting the most accurate ML/DL model – although the ranker helps to reduce the computation load for other system components, it is crucial to balance the trade off between inference accuracy and cost, with respect to the end-to-end system performance. To evaluate the benefits of the learned ranker as a building block for learning-augmented systems, we have integrated it into popular regex matching engines: PCRE [3], PCRE-JIT [4], and RE2 [5]. We benchmark with two publicly available rulesets: ModSecurity CRS [2] and Snort [6]. Empirical results show that the learning-augmented design reduces the rule-matching system latency by as much as 78.81%; in particular, the learned rule ranker can achieve a top-5 ranking accuracy at least 96.16%, and this reduces the average number of per-input regex matching invocations by as much as 98.54%.

2 System Overview

Figure 1 illustrates the learning-augmented design of rule-matching systems. The rule ranker exploits the fact that many rulesets do not fix a mandatory ordering of rules, and it dynamically re-orders rules according to how likely they would match the given input. If the ranker successfully prioritizes the matching rule among the top N candidates, then the regex matching engine can effectively early-terminate after checking at most N rules. The figure also illustrates that the learned rule ranker can complement many existing optimization solutions. First, there is a string-matching pre-filter that first removes inputs unlikely to match any rule [7, 12]. Second, there are efforts on reducing the regex matching engine latency, e.g., PCRE's just-in-time compilation [4] and hardware-based regex acceleration [20, 25].

The rule ranker can take different realizations and ML/DL models. While ranking accuracy is a primary consideration in designing the ranker, achieving high accuracy typically comes with the cost of computation overhead and latency. This trade-off is crucial, as each input incurs the inference cost. Therefore, it is possible that a ranker does not speed up the overall system performance – in the context of rule-matching systems, these worst cases happen when a given input triggers a large amount of rule processing. Possible reasons include ***(1)*** the string-matching pre-filter fails to first remove unlikely inputs, or ***(2)*** the ranker fails to optimally prioritize rules.

2.1 Strawman Solutions for Rule Ranker

Static and heuristics-based solutions. If the overall system workload exhibits a long tail in the rule hit distribution (e.g., some rules account for a majority of matches), then both static and heuristics-based solutions can be effective. In particular, for cases where system workloads are assumed to rarely exhibit temporal dynamics, system operators can sort rules by statistically counting the number of rule hits in historical logs. Otherwise, heuristics such as least recently used (LRU) and least frequently used (LFU) can be used to improve the adaptability to sporadically temporal dynamics.

While both strawman solutions are simple, they can be sub-optimal due to the following reasons. First, while inputs are independent, these solutions rank rules based on the historical hit distribution, rather than any features of the current input. Second, they are suitable only for scenarios with known or long-tailed rule hit distributions. For cases and systems where different inputs can match different rules, LRU and LFU might not work well if the principle of locality does not hold.

Classification-based solutions. The rule-matching problem can be formulated as a multi-class classification problem in the machine learning domain. Specifically, assuming each rule is one class, we aim to predictively classify an input and rank rules by the likelihood score of each class. One widely-used non-DL classification technique is the logistic regression (LR). While being used traditionally for single-class classification, LR can be extended for multi-class classification through the one-vs-rest strategy. Unlike linear regression and support vector machine (SVM), LR is able to output probabilistic values, rather than binary answers. Probabilistic values are useful in comparing the relative likelihood of rules in a ruleset.

As a strawman solution, LR can be sub-optimal due to the following reasons. First, since the one-vs-rest-strategy [11] requires one model for each rule, a ruleset with r rules would result in r LR models. In addition to the training cost, each input effectively forces inferences over all r models. Second, since LR commonly targets linearly separable datasets, it is inadequate to model the space of matching inputs for rules of complicated regex conditions. §5 compares LR with DL-based solutions.

3 Learned Rule Ranker

Recent advances from deep learning communities have driven the availability of off-the-shelf DL models such as the popular

fully connected Deep Neural Networks (DNN) and Recurrent Neural Networks (RNN). DL models have the following advantages in the context of realizing learned rankers for modern systems. First, DL models can model complex non-linear datasets (e.g., rules with complicated conditions in our case). Second, DL models have hyper-parameters (e.g., number of hidden layers and neurons) that can easily be tuned to optimize the trade off between model accuracy and inference latency. Third, although DL models have been known to require a large amount of training data, rule-matching system inputs can be randomly generated and cheaply labeled.

Deploying DL models for learned rule ranker involves the following considerations and customizations.

Model selection. Given that inputs are strings, we consider the use of both DNN models (for their simplicity) and RNN models (for their ability to handle an arbitrary length of texts). As §5 shows, DNN typically has a lower inference latency, and RNN typically has a higher accuracy in prioritizing the matching rule among the top-*N* candidates. However, we argue that model selection goes beyond simply selecting the most accurate model configuration. Being a system building block, the learned ranker design must consider how the inference accuracy and costs would impact the end-to-end system performance. We illustrate this consideration with Figure 1 – if the regex matching engine is fast, having a relatively inaccurate rule ranker might be a reasonable design, especially if the overhead of checking one unnecessary rule is lower than the inference overhead of more accurate rankers. At the same time, a relatively inaccurate rule ranker might hurt the overall system performance, especially if the reduction in the amount of unnecessary rule checking does not adequately compensate the inference overhead.

Model inputs and outputs. The input layer of a neural network takes in a vector of real numbers. Since inputs in our case are a string of characters, they go through the process of word embedding to convert individual characters into 8-bit numbers in ASCII encoding. Furthermore, we note that DNN needs to take the entire input string at once, which forces the DNN input layer size to be at least as large as the input string. While the maximum input string length needs to be decided beforehand, system operators usually have statistics on the typical system workload. If an input string is shorter than the maximum length, we pad "0" at the end of the vectorized input. On the other hand, since RNN can take the input string in chunks, it can handle inputs of arbitrary length.

The output layer has a set of neurons where each neuron corresponds to a particular rule. Each neuron outputs a number between 0 and 1 representing the classification probability. We use these outputs to rank rules.

Input Generator. In addition to real-world traces of rule-matching system inputs, ranker training can happen with artificially generated matching/unmatching inputs. One advantage that the input generator offers is the large quantity of training data necessary for training DL models. To generate training inputs for a rule, our input generator runs Xeger [21] and Exrex [22], which are popular Python libraries for generating random strings from a given regex. Then, we randomly repeatedly choose S random characters from each of these generated inputs, and replace them with random characters. These mutated strings are then classified as either the `matching` and `unmatching`, by running the regex matching engine. The value of S is a crucial parameter – a larger S produces a nearly random unmatching string, and a smaller S changes only a few characters to simulate "near-miss" cases in the real world.

Training. With training inputs collected in the real world or generated by the input generator, we follow the popular training method of backward propagation with gradient descent. Since training data are labeled, the training is effectively a supervised learning. We use batch training, and each batch contains one input for each rule and one unmatching input. In addition, we train the DL model with 1,000 epochs, we use ReLu as the activation function at hidden layers and we use softmax at the output layer for outputting classification probabilities.

4 Implementation

We implement our learned ranker in Python 3.6 and TensorFlow 1.10.0. Our current implementation consists of ∼1,400 lines of Python code. In order to optimize the performance of TensorFlow on a CPU, we recompile the library with SSE 4.2, AVX and FMA instructions. We also enable just-in-time compilation for TensorFlow graphs.

To expose the target rule-matching system for the purpose of labelling inputs, we write a client stub. The client stub receives input strings from our input generator, and calls the target system's API. The communication between the input generator and the client stub happens over HTTP with messages in the JSON format.

5 Evaluation

Our major results include – ***(1)*** a learned rule ranker can reduce the average number of per-input regex matching invocations by as much as 98.54%, with a top-5 ranking accuracy of at least 96.16%. ***(2)*** Factoring in the rule ranker inference overhead, the learning-augmented design reduces the rule matching latency by as much as 78.81%. ***(3)*** We demonstrate that the ranking model design should consider a global optimization strategy, as having the most accurate model does not necessarily benefit the end-to-end system performance.

5.1 Methodology

Rulesets. While a learned rule ranker is not limited to security-related scenarios, two popular rulesets that are publicly available are ModSecurity CRS v3.0 [2] and Snort v3.0 [6]. The former is a web application firewall module, and the latter is a network-based intrusion detection system. We are interested in rules with complicated regular expressions with metacharacters, rather than simple string matching – our RS_{CRS} ruleset consists of 69 regex rules ranging from 20 to 3447 characters, and RS_{Snort} ruleset consists of 196 regex rules ranging from 21 to 243 characters. We note that rules can have matching criteria on multiple input fields, e.g., `ARGS`, `ARGS_NAMES`, `REQUEST_COOKIES`, and `REQUEST_COOKIES_NAMES` in CRS, and `HTTP_header`, `HTTP_uri`, and `HTTP_method` in Snort.

Workload datasets. Our experiments are based on following rule-matching system workloads: ***(1)*** the public ECML data set, WL_{ECML} [1], and ***(2)*** artificial data sets generated from the CRS and Snort rulesets, WL_{CRS} and WL_{Snort}. The former is primarily used as testing dataset. The latter can drive training and testing, by separately generating multiple sets of inputs.

For the artificial data sets, an input generator (c.f. §3) outputs random matching and unmatching strings, with respect to the given regex pattern. Unmatching strings allow us to test rules that require only some of the specified fields to match. The tool can generate a balanced workload to simulate the worst case where all rules are likely to be hit.

Testbed environment. We run popular rule-matching engines including PCRE [3], PCRE-JIT [4], and RE2 [5]. PCRE is the most widely used open-source regex matching engine, and the JIT optimization minimizes unnecessary parsing of the internal bytecode representation, especially the matching engine can contain many unused code branches from `if` and `switch` statements. RE2 is a fast and thread-friendly regex matching engine. We use Python and carry out experiments on a Ubuntu-based Azure VM with access to 8 cores of Intel Xeon E5-2673 running at 2.4 GHz and 3 GB of RAM.

5.2 Rule Ranking Accuracy

The primary goal of the learned rule ranker is to minimize unnecessary regex rule matching, and the system performance gain depends on its effectiveness in correctly prioritizing the matching rule as a top candidate. We quantify the effectiveness by the top-N accuracy, or the probability that the rule ranker successfully prioritizes the matching rule to be one of the first N rules to check. This subsection evaluates the different factors of the top-N accuracy.

Impacts of model selection. One factor that can impact the rule ranker's top-N accuracy is the learning model, and we empirically evaluate rule rankers implemented by DNN models (with two hidden layers of 128, 256, and 512 neurons),

Model	Top-1	Top-3	Top-5	Latency (μs)
DNN(128)	81.85%	94.39%	97.05%	11.65
DNN(256)	83.37%	95.18%	97.45%	14.68
DNN(512)	83.72%	95.61%	97.52%	21.44
RNN(128)	89.44%	97.51%	98.82%	33.43
RNN(64)	92.98%	98.55%	99.30%	39.88
RNN(32)	95.02%	99.23%	99.71%	48.02
LR	67.69%	82.89%	88.18%	48.25

(a) RS_{CRS}

Model	Top-1	Top-3	Top-5	Latency (μs)
DNN(128)	80.08%	93.34%	96.16%	11.75
DNN(256)	83.14%	94.69%	97.27%	15.42
DNN(512)	84.45%	95.34%	97.41%	22.44
RNN(128)	85.59%	96.88%	98.26%	41.62
RNN(64)	91.33%	98.19%	99.18%	46.21
RNN(32)	94.45%	99.22%	99.63%	56.21
LR	83.83%	93.29%	95.65%	93.19

(b) RS_{Snort}

Table 1: One factor that impacts rule ranking accuracy is the learning model selection: DNN (with two hidden layers of 128, 256, and 512 neurons), RNN (with input chunk size of 32, 64, and 128 characters), and logistic regression (LR). Results illustrate the trade off between top-N accuracies and ranking latency.

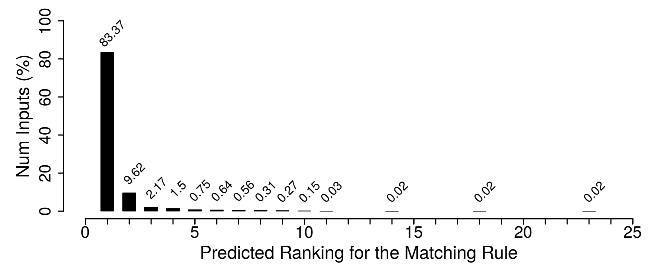


Figure 2: Distribution of the predicted ranking for matching rules. For 83.37% of inputs, the rule ranker is able to prioritize the matching rule as the first candidate.

RNN models (with input chunk size of 32, 64, and 128 characters), and logistic regression (LR). We train each model with 100,000 inputs from WL_{CRS} and WL_{Snort}. Table 1a and 1b show empirical measurements for WL_{CRS} and WL_{Snort}, respectively. We make the following observations. RNN and LR have the highest and lowest top-N accuracies, respectively. While DNN (512) exhibits a 2.19% lower top-5 accuracy than RNN (32), it is $\sim 2.24\times$ faster. This trade-off suggests that simply using top-N accuracies as the selection metric might not benefit the entire system, and we further discuss how the trade-off between top-N accuracies and inference costs impacts the end-to-end rule matching throughput in §5.3.

Next, we look at inputs where the rule ranker fails to properly prioritize rules. Since these inputs require the regex matching engine to process more rules, they have a higher rule matching latency. Figure 2 illustrates the distribution of the predicted ranking for matching rules in the case of

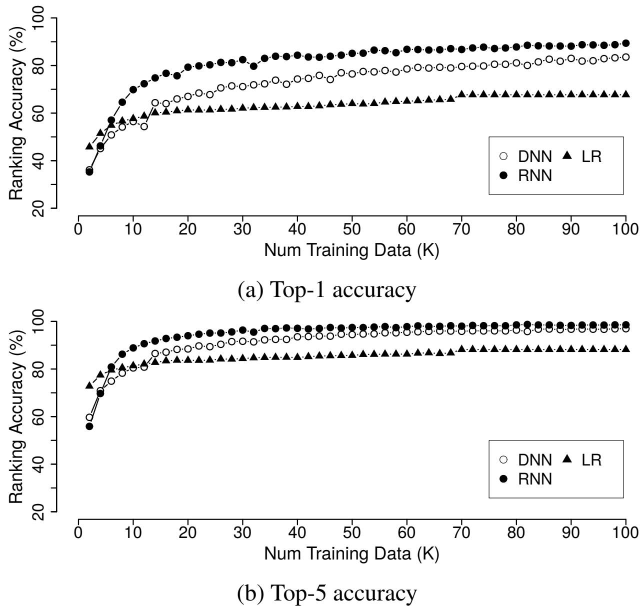


Figure 3: Increase in ranking accuracy in terms of training data size, in the case of RS_{CRS}.

DNN(256). For 83.37% of inputs, the rule ranker is able to prioritize the matching rule as the first candidate. For 2.55% of inputs, it fails to prioritize the matching rule as a top-5 candidate. Interestingly, most of these inputs are relatively short, and the excessive padding might cause the ranker to infer those inputs incorrectly.

Impacts of training dataset size. Another factor that can impact the rule ranker's top-*N* accuracy is the amount of training data. Figure 3 shows how the accuracy increases for different DL/ML models in the case of RS_{CRS}, and we evaluate the accuracy by testing 1,000 randomly generated inputs (after each training iteration with 2,000 generated inputs). We note that our models generally start to converge after being trained with ~90,000 inputs.

Impacts from imbalanced workload distributions. We acknowledge that, if the system operator has a complete prior knowledge of the workload distributions, it is possible to hardcode a static rule checking order. One case where this is particularly useful is the long-tailed rule hit distribution. In other words, the workload is imbalanced such that a majority of inputs match only a subset of the ruleset. Compared to WL_{CRS} and WL_{Snort}, the `ECML` dataset is relatively imbalanced. And, empirical results show that static order can significantly reduce the number of rules that the regex matching engine needs to process for WL_{ECML}.

However, given that the learned rule ranker is able to prioritize rules with each input's features, it can actually achieve a larger reduction – compared to static ordering, the DNN-based rule ranker reduces by 88.70% and 56.04% for RS_{CRS} and RS_{Snort} in the case of WL_{ECML}, respectively.

Regex engine	Without rule ranker	With rule ranker	Reduction
PCRE	1878.79 μsec	404.36 μsec	78.47%
PCRE-JIT	773.82 μsec	185.65 μsec	78.81%
RE2	206.01 μsec	55.15 μsec	73.22%

Table 2: Latency for matching one input with DNN(256)-based ranking model, on the RS_{CRS} ruleset.

Ruleset	No rule ranker	Rule ranker	Reduction
RS_{CRS}	22.38	1.68	92.49%
RS_{Snort}	91.56	1.34	98.54%

Table 3: Average number of regex rules that the regex matching engine needs to process for each input.

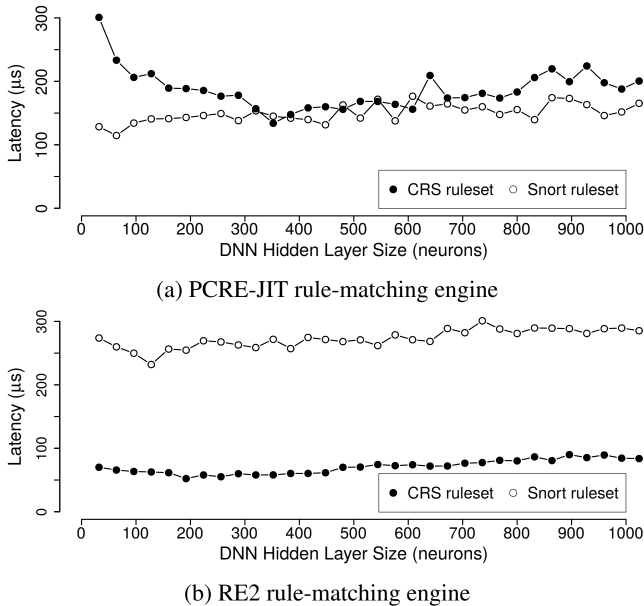
5.3 Rule Matching Latency Reduction

We next evaluate the rule matching latency reduction from the learned rule ranker, and we integrate the learned rule ranker into the PCRE, PCRE-JIT, and RE2 engines.

Table 2 shows the latency reduction for processing one input with different regex matching engines, on the RS_{CRS} ruleset. Each experiment runs 10,000 different inputs to avoid measurement noise. By fixing the ranking model, we observe that the reduction varies with different regex matching engines – the reduction ranges from 73.22% to 78.47%. This reduction in latency is correlated with the reduction in the number of rules that a regex matching engine needs to process. Table 3 shows that, for RS_{CRS}, the average number of regex matching invocations is 1.68, which is a 92.49% reduction from 22.38. The reduction in the case of RS_{Snort} is 98.54%. We note that, since both WL_{CRS} and WL_{Snort} are fairly balanced workloads (i.e., the number of matching inputs for each rule is roughly the same), per-input rule prioritization is key to reduction. And, the learned ranker exhibits a higher gain in cases where the overhead of processing an unnecessary rule is higher.

Finally, we highlight that the design of learning-augmented systems should consider the trade off between the learning's interference costs and the system's global performance gain. Figure 4 illustrates results from a DNN-based ruler ranker. Although having larger hidden layers improves DNN model accuracy, it does not necessarily benefit the end-to-end matching latency. This is due to the fact that, not only does an accurate model reduce unnecessary rule checking, but it also imposes higher inference costs to the end-to-end system performance. Unfortunately, the optimal model configuration depends on the execution environment and system configurations – in our case, the selection of regex matching engine and ruleset. For instance, Figure 4 shows that the optimal DNN hidden layer size is ~192 neurons for RE2 with RS_{CRS} ruleset, ~128 neurons for RE2 with RS_{Snort} ruleset, ~256 neurons for PCRE-JIT with RS_{CRS} ruleset, and ~64 neurons for PCRE with RS_{Snort} ruleset.

Discussion. We note that certain ruleset characteristics are also factors that potentially impact the overall rule matching



(a) PCRE-JIT rule-matching engine

(b) RE2 rule-matching engine

Figure 4: This figure illustrates that, although having larger hidden layers improves DNN model accuracy, it does not necessarily benefit the end-to-end matching latency. The reason behind this observation is the correlation between model accuracy and inference costs.

performance. One prominent example is the number of rules in the ruleset. Specifically, as the ruleset becomes larger, successfully prioritizing the matching rule means more rules can be skipped. On the other hand, a larger ruleset requires more complicated ML/DL models, which can impose additional overhead on the system performance. Given the lack of tools to automatically generate rulesets of different characteristics, our experiments in this section are based on two public rulesets, RS_{CRS} and RS_{Snort}. We leave the evaluation of ruleset characteristics as future work.

6 Related Work

Accelerating rule matching. The realization of regular expression patterns as finite state automata was first proposed by Kleene et al. [14] in the 1950s, and NFA and DFA have been widely used to formalize the description of regex patterns. Since most rule-based systems (such as traffic detection, data retrieving, and even DNA sequence matching) rely on regular expression patterns for condition matching, there have been efforts in speeding up regex pattern matching.

Some efforts focus on software optimizations for finite state automaton. PCRE-JIT [4] uses the just-in-time (JIT) library to minimize unnecessary parsing of the internal bytecode representation. Recently, Choi et al. proposed DFC [12], a memory-efficient and cache-friendly data structure that minimizes CPU stalls to maximize instruction-level parallelism. Kumar et al. [16] proposed a representation of regular expression patterns called Delayed Input DFA (D^2FA), which reduces the space requirement as compared to DFA. Further efforts build upon D^2FA [10, 18], and compress states and transitions. Finally, some efforts leverage hardware capabilities to speed up automaton – Mitra et al. [20] and Yuan et al. [26] explored the use of FPGAs and GPUs, respectively.

Being a system building block, the learned rule ranker should complement many existing optimizations.

Learning-augmented systems. Auto-tuning system parameters is a popular scenario for learning-augmented systems. OtterTune [8] is a database optimization tool. It uses a combination of supervised and unsupervised machine learning methods to reduce the parameter dimension, characterize observed workloads, and recommend configurations. CherryPick [9] demonstrates the potential of using Bayesian optimization and Gaussian process in predicting the best-performing cloud configuration for a given machine learning computation workload. Metis [17] addresses challenges that systems introduce to hinder the tuning robustness.

Furthermore, the networking community [24] has been applying ML and DL techniques to traffic prediction, traffic classification, resource management, network adaption, etc. To improve the QoS metric of video streaming, Mao et al. [19] used reinforcement learning in the rate adapting mechanism to continuously and adaptively adjust the streaming bit rate.

Finally, Kraska et al. [15] proposed the learned index to replace common indexes in databases. The learned index formulates the problem of database indexes as a DL predictive problem. It offers similar semantic guarantees, and a significant improvement in speed and memory efficiency.

Building on the success of these efforts, we explore whether the learned ranker can be a building block of learning-augmented systems.

7 Conclusion

This paper explores the potential and feasibility of learned ranker as a building block for learning-augmented systems. Particularly, we use rule-matching systems as a concrete scenario. To evaluate the benefits of the learned ranker, we have integrated it into popular regex matching engines. As future work, we plan to study challenges in training learned rankers, and apply learned rankers to other system scenarios.

Acknowledgments

We thank anonymous reviewers and our shepherd, Dr. Julia Lawall, for their constructive feedback and suggestions. This work is partly supported by the Youth Innovation Promotion Association of CAS and the National Natural Science Foundation of China (No.61772485 and No.61432016).

References

[1] ECML/PKDD 2007 Discovery Challenge - Analyzing Web Traffic. http://www.lirmm.fr/pkdd2007-challenge.

[2] ModSecurity - Open Source Web Application Firewall. https://modsecurity.org.

[3] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org.

[4] PCRE JIT. http://www.pcre.org/original/doc/html/pcrejit.html.

[5] RE2. https://github.com/google/re2.

[6] Snort. https://www.snort.org.

[7] Hyperscan: Turbo Boosting Regular Expression Matching for Network Security Applications. In *NSDI (Operational Systems Track)*. USENIX, 2019.

[8] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 2017.

[9] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX, 2017.

[10] Michela Becchi and Patrick Crowley. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *ANCS*. ACM, 2007.

[11] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[12] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. DFC: Accelerating String Pattern Matching for Network Applications. In *NSDI*, 2016.

[13] Russ Cox. Regular Expression Matching Can Be Simple And Fast (But Is Slow in Java, Perl, PHP, Python, Ruby, ...). http://swtch.com/~rsc/regexp/regexp1.html, 2007.

[14] Stephen Cole Kleene. Representation of Events in Nerve Nets and Finite Automata. Technical report, United States Air Force, 1951.

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*. ACM, 2018.

[16] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *ACM SIGCOMM Computer Communication Review*. ACM, 2006.

[17] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC*. USENIX, 2018.

[18] Alex X Liu and Eric Torng. An Overlay Automata Approach to Regular Expression Matching. In *INFOCOM*. IEEE, 2014.

[19] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *SIGCOMM*. ACM, 2017.

[20] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for Accelerating Snort IDS. In *ANCS*. ACM, 2007.

[21] Colm O'Connor. Xeger. https://pypi.org/project/xeger/, 2018.

[22] Adam Tauber. Exrex. https://pypi.org/project/exrex/, 2018.

[23] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytic. In *NSDI*. USENIX, 2016.

[24] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 2018.

[25] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed Regular Expression Matching Engine Using Multi-character NFA. In *FPL*. IEEE, 2008.

[26] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA Implementation for Memory-efficient High-speed Regular Expression Matching. In *PPoPP*, 2012.

MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving

Chengliang Zhang Minchen Yu Wei Wang
HKUST
{czhangbn, myuaj, weiwa}@cse.ust.hk

Feng Yan
University of Nevada, Reno
fyan@unr.edu

Abstract

The advances of Machine Learning (ML) have sparked a growing demand of ML-as-a-Service: developers train ML models and publish them in the cloud as online services to provide low-latency inference at scale. The key challenge of ML model serving is to meet the response-time Service-Level Objectives (SLOs) of inference workloads while minimizing the serving cost. In this paper, we tackle the dual challenge of SLO compliance and cost effectiveness with MArk (Model Ark), a general-purpose inference serving system built in Amazon Web Services (AWS). MArk employs three design choices tailor-made for inference workload. First, MArk dynamically batches requests and opportunistically serves them using expensive hardware accelerators (e.g., GPU) for improved performance-cost ratio. Second, instead of relying on feedback control scaling or over-provisioning to serve dynamic workload, which can be too slow or too expensive for inference serving, MArk employs predictive autoscaling to hide the provisioning latency at low cost. Third, given the stateless nature of inference serving, MArk exploits the flexible, yet costly serverless instances to cover the occasional load spikes that are hard to predict. We evaluated the performance of MArk using several state-of-the-art ML models trained in popular frameworks including TensorFlow, MXNet, and Keras. Compared with the premier industrial ML serving platform SageMaker, MArk reduces the serving cost up to $7.8\times$ while achieving even better latency performance.

1 Introduction

Driven by the sustained advances of Machine Learning (ML), the past few years have seen a surging demand of ML-as-a-Service (MLaaS). A typical workflow of MLaaS covers the two phases of ML in the cloud: *training* and *inference*. In the training phase, developers build ML models from the training dataset using an array of ML frameworks. Efficient training in cloud environments has been well explored in the recent work [43, 56, 75]. In the inference phase, the trained models are published as *online services* in data center or cloud and can be queried by end users with new input. The service makes prediction decisions (inference) for a given input using the trained model [30] (e.g., recognizing human faces in a given photo), and returns the inference results to the querier.

Unlike training which runs offline and may take hours to days to complete, inference must be performed in *real-time* on dynamic queries with stringent latency requirements (e.g., tens to hundreds of milliseconds per query). These requirements are often specified as the *response-time Service-Level Objectives* (SLOs) [41], such as at least 98% of inference queries must be served in 200 ms. Failing to comply with the SLOs results in compromised quality of service or even financial loss, e.g., end users will not be charged for queries not responded in time. Therefore, an ML model serving system should strive to meet the target SLOs while minimizing the cost of provisioning the serving instances in the cloud.

However, achieving these two objectives can be challenging. Cloud providers like Amazon [11], Google [37], and Microsoft [52] offer a rich selection of service provisioning options, ranging from VMs and containers to the emerging serverless functions. For each provisioning option, there is a large configuration space (e.g., CPU, memory, and hardware accelerators) coupled with diverse pricing models offering tradeoffs between service guarantees and cost savings (e.g., on-demand and spot instances [17]). A key challenge of provisioning model serving in the cloud is: how should a serving system choose from a bewildering array of cloud services to provide low-latency, cost-effective inference at scale?

Unfortunately, there is no general guideline given by the cloud providers, nor has it been studied in the prior work [10, 25, 42, 45, 58, 59, 63, 70] which mainly targets at general workload. To bridge this gap, we perform extensive measurement studies of inference serving in AWS [11] and Google Cloud [37] by means of VMs (IaaS), containers (CaaS), and serverless functions (FaaS). We briefly summarize three key findings as follows.

First, our measurements suggest that among the three options, IaaS offers the best performance-cost ratio for inference serving, but it incurs long instance provisioning latency and is hence unable to quickly adapt to the changing workload. CaaS suffers from a similar problem as IaaS (though less

severe) with worse performance-cost ratio. Compared to IaaS and CaaS, FaaS scales much faster but is the most expensive.

Second, inference serving can gain significant benefits from *batching* when performed using costly hardware accelerators (e.g., GPU and TPU). Nevertheless, the benefits are not always guaranteed but critically depend on the batch size control knobs and their interactions with query arrivals: when there is not enough load, serving inference queries using GPUs is not economically justified. Therefore, a serving system should judiciously determine when to *scale up* from CPU to GPU instances and how to perform batching over GPUs.

Third, ML inference usually performs *stateless* computations. This opens up an opportunity of using serverless functions as a *handover service* when the system is provisioning new instances for scaling up/out. Also, many ML models, especially deep learning, have *deterministic inference time* [41, 74]—they take fixed-size input vectors and have input-independent control flows. This also brings an opportunity for better resource planing and latency control.

Motivated by these observations, in this paper, we propose MArk (Model Ark), a low-latency, cost-effective inference serving system in the public cloud. MArk takes use of the unique characteristics of ML model serving while also addressing the distinctive challenges posed by it. In particular, MArk allows developers to specify the target SLOs through common APIs. To attain high performance-cost ratio, it uses IaaS as the primary means of provisioning while employing FaaS to quickly fill the service gap when the system is undergoing horizontal/vertical scaling. MArk uses *predictive scaling* to hide the instance provisioning latency in IaaS. Unpredicted load spikes are covered by serverless functions to reduce over-provisioning. Based on the predicted workload, MArk opportunistically uses costly GPU instances to serve *batched queries* for improved performance-cost ratio. To further bring down the cost, MArk also supports the use of the discounted, yet *interruptible instances* (e.g., spot instances) with an interruption-tolerant mechanism that uses transient servers to handle instance interruptions at low cost.

We have prototyped MArk as a general-purpose serving platform in AWS [11] with pluggable backend model servers supporting a range of ML frameworks such as Tensorflow Serving [55], MXNet Model Server [24], and customized Keras [29] server with Theano [26] backend. We have evaluated MArk on AWS using several state-of-the-art ML models for image recognition, language modeling, and machine translation: Inception-V3 [67], NASNet [76], LSTM-ptb [51], and OpenNMT [47]. The results show that MArk yields up to 7.8× cost reduction while achieving comparable or even better latency compared to the state-of-the-practice solution SageMaker [13], and also comply with the predefined SLO requirements. MArk is open-sourced for public access.[1]

[1] https://github.com/marcoszh/MArk-Project

2 Background and Related Work

In this section, we survey related work on model serving systems and autoscaling techniques. We also provide background information on cloud services and their pricing models.

2.1 Machine Learning Model Serving

A wide array of ML inference serving systems have been proposed to facilitate model deployment [7, 8, 24, 30, 55, 72]. These systems place the trained models in *containers* and handle model inference requests through REST APIs. For example, systems like Clipper [30], Rafiki [72], and MXNet Model Server [24] host each model in a separate Docker [4] container to ensure process isolation; TensorFlow Serving [55] deploy models as *servables*, which are executed as black box containers and can also be used for version management. In order to provide low-latency inference, these systems employ a number of model-agnostic optimizations such as batching, buffering, and caching [30]. The recently proposed *white box* model serving [49] enables model-specific optimizations with fine-grained resource sharing and parameter re-use.

However, existing inference serving systems mainly focus on streamlining model deployment in server machines, without addressing the scalability and cost minimization issues for model serving on the public cloud. Microsoft's Swayam [41] is among a few inference serving systems that focus on infrastructure scalability and resource efficiency. Yet, Swayam is a proprietary system for model deployment in Microsoft's private MLaaS clusters, where the cloud provisioning options (e.g., IaaS, CaaS, FaaS) and their pricing models are not relevant. Amazon's SageMaker [13] offers scalable model serving over EC2 [1] instances. However, it only supports IaaS provisioning and requires manual specification of the provisioning instances. SageMaker is also *agnostic* to the response-time SLOs and serves inference queries in a best-effort manner. In contrast, MArk meets SLOs at low cost by choosing from a complex selection of provisioning services in AWS [11].

2.2 Autoscaling Dynamic Workload in Cloud

There is a large body of work on autoscaling dynamic workload for general web services hosted in the cloud. We refer to [59] for an extensive survey of this topic and compare some related work with MArk in Table 1. In general, there are two scaling approaches used to serve dynamic workload.

Feedback control scaling. This approach monitors hosted applications and *reactively* adjusts resource provisioning based on the monitored metrics (e.g., utilization, throughput, and latency). Feedback control scaling is adopted in many industrial serving platforms to autoscale dynamic workload, e.g., SageMaker in AWS [12, 13] and Kubernetes in Google Cloud [38, 39]. These systems perform scaling following some customized rules such as "adding 2 instances if CPU

Table 1: A comparison of MArk and existing work on autoscaling dynamic workload in the cloud.

Autoscaler	Scaling approach	Means of Provisioning	SLO-aware	Heterogeneous instances	Interruptible instances	Hardware accelerators
MBRP [33]	Feedback control	Private cluster	✓	✓	×	×
Ali-Eldin et al. [9]	Predictive	IaaS	×	×	×	×
Barrett et al. [25]	Predictive	IaaS	×	×	×	×
Urgaonkar et al. [70]	Predictive	IaaS	✓	×	×	×
Han et al. [42]	Predictive	IaaS	✓	×	×	×
Qu et al. [58]	Feedback control	IaaS	×	✓	✓	×
SpotCheck [63]	–	IaaS	×	✓	✓	×
He et al. [45]	–	IaaS	×	✓	✓	×
Swayam [41]	Predictive	Private cluster	✓	×	–	×
SageMaker [13]	Feedback control	IaaS	×	×	×	✓
MArk	Predictive	IaaS and FaaS	✓	✓	✓	✓

utilization reaches 70%," or tracking a target such as "maintaining 100 queries per minute per instance" [15].

Feedback control scaling makes no prediction about the future and is easy to implement. However, owing to its reactive nature, it incurs long instance provisioning delay when used to serve changing workload [59]. Over-provisioning is therefore needed in case of load spikes. For example, SageMaker recommends to start with 100% over-provisioning and adjust thereafter [16]. As ML model serving is often compute-intensive and requires costly CPU/GPU instances, solely relying on over-provisioning is economically not viable.

Predictive scaling. This approach makes predictions about the future workload, based on which it *proactively* autoscales the serving instances to reduce over-provisioning. Predictive scaling has been widely employed to serve general workload (e.g., web services and VM demands) using a number of time-series based prediction algorithms, such as linear regression [27], autoregressive models [34, 61], and neural networks [19, 53, 57, 65]. Predictive scaling is often complemented with feedback control scaling, where the two approaches operate at different time scales [42,70]. For example, predictive scaling can be used for resource planning at the time scale of hours or days, while reactive provisioning operates in minutes to respond to flash crowds or unexpected deviations from long-term behaviors [70].

However, due to the mismatch of target workload, existing predictive autoscalers do not work well for ML model serving. As summarized in Table 1, they only consider provisioning over homogeneous instances in IaaS [9, 25, 42, 70]. They also do not support hardware accelerators (e.g., GPUs) and cheaper, yet interruptible instances (e.g., spot servers), hence missing opportunities of cutting provisioning cost. In addition, many predictive autoscalers are unaware of the response-time SLOs and only provide best-effort services [9, 25].

2.3 Cloud Provisioning Services

Compared with private clusters, model serving in public clouds is more complex. Leading cloud platforms such as AWS [11], Google Cloud [37], and Microsoft Azure [52] offer a variety of provisioning services that can be used for model serving. We briefly review these services, with a main focus on AWS.

Infrastructure-as-a-Service (IaaS). With IaaS, cloud customers run virtual instances (VMs) of various configurations in terms of vCPUs, memory, storage, network, and accelerators (e.g., GPU, TPU, and FPGA). Customers can then configure and deploy ML model serving softwares [24,30,68] on running instances to serve model inference requests.

IaaS cloud provides flexible pricing options to allow customers to choose between service guarantees and cost savings. Taking Amazon EC2 [1] as an example, customers can run instances *on-demand* and pay for compute capacity by per hour or per second depending on the instance types. Alternatively, customers can run *spot instances* at steep discounts compared to the on-demand price, under the condition that a running spot instance can be *interrupted indefinitely* [17]. EC2 also allows customers to *reserve* an instance in a long term by making an upfront payment [21]. During the reservation period, the instance usage is subject to a heavy discount compared to the on-demand price. All three IaaS pricing options are also available in Google Cloud [37].

Container-as-a-Service (CaaS). With CaaS, customers encapsulate services and implementations in containers (e.g., Docker images [4]), and run containers with specified resource configurations in the cloud, e.g., Amazon ECS [2] and Google Kubernetes Engine [6]. Compared with IaaS, CaaS simplifies software configurations and deployment without the complexity of maintaining the server infrastructure. In Amazon ECS, users pay for the container capacity by per second, where the pricing is based on requested vCPU cores and memory.

Function-as-a-Service (FaaS). With FaaS, customers run applications as *serverless functions* in the cloud without provisioning or managing servers, e.g., AWS Lambda [3] and Google Cloud Functions [5]. In Lambda, customers can only specify the memory allocation for an instance, and pay for the total number of requests and the duration of compute time [3]. FaaS is particularly suitable for *stateless computations* and

Table 2: Cost ($) and average latency ($t$) of serving 1 million requests of three ML models in AWS. We choose c5.large EC2 instance (2 vCPUs and 4GB memory) as it is the most cost-effective. Each ECS container is allocated the same vCPUs and memory as c5.large; each Lambda instance has 3GB memory to achieve comparable latency with c5.large.

ML Model	EC2		ECS		Lambda	
	$	t (ms)	$	t (ms)	$	t (ms)
Inception-v3	5.0	210	9.17	217	19.0	380
Inception-ResNet	9.3	398	16.4	411	39.3	785
OpenNMT-ende	51.5	2180	96.3	2280	155	3100

has recently been used to provision ML model serving [69].

Given a complex selection of provisioning options in the public cloud, which one should be used for ML model serving? We answer this question in the next section.

3 Characterizing Model Serving in the Cloud

In this section, we characterize ML serving performance with IaaS, CaaS, and FaaS as well as their configuration space. Our characterizations are mainly based on AWS [11] (§3.1-3.4), a leading cloud platform offering the most diversified service options. We validate the major results in Google Cloud [37] where possible (§3.5).

3.1 What service to use: IaaS, CaaS, or FaaS?

We choose three representative ML models, Inception-v3 [67], Inception-ResNet [66], and OpenNMT-ende [47], for common prediction tasks such as image classification and machine translation, and evaluate their peak inference performance with TensorFlow Serving [55]. Table 2 summarizes the cost and average latency of serving 1 million requests using AWS EC2 (IaaS), ECS (CaaS), and Lambda (FaaS), respectively.[2]

IaaS vs. CaaS. In EC2 [1], customers can choose among predefined instance types with fixed vCPU and memory allocation. In Table 2, we choose the cheapest compute-optimized instance c5.large as the reference, since it is proven to be the most cost-effective one in §3.3. AWS's container service ECS [2], on the other hand, lets users choose the number of vCPUs they want. We allocate each container with 2 vCPUs to match the capacity of c5.large, and with the minimum memory allowed. Compared with c5.large, the ECS container has similar serving latency but is more expensive.

FaaS. As for the serverless computing service Lambda [3], the pricing is per-request based, and the cost per request depends on the resource allocation and runtime of the request. Customers specify memory allocation in Lambda, and CPU resource is allocated proportionally to memory [14]. For a fair comparison, we compare the Lambda cost of serving the same amount of requests c5.large can serve in an hour, with the maximum memory allocated for best performance. The cost is significantly higher, and the latency is longer, too.

Scalability. EC2 has long provisioning overhead (e.g., several minutes), because additional time is needed to load and set up large ML model serving atop standard overhead, as Microsoft suggests with their production traces [41]. The overhead makes it challenging to accommodate demand surge without high margin of over-provisioning. The high launching overhead also penalizes frequent provisioning and deprovisioning, since customers are billed during the instance launching period as well. Similar to EC2, ECS also needs dozens of seconds of provisioning overhead. Lambda, on the contrary, is able to spawn thousands of new ML inference instances in less than a few seconds, and once an instance is ready, it can continuously serve requests without incurring additional overhead [48]. The cold start overhead of Lambda can be amortized by warming up [48]. Compared with EC2 and Lambda, ECS shows no obvious advantage.

Summary. A natural question is that can we exploit the cost-effectiveness of IaaS service while also taking advantage of the high scalability of FaaS? Conventional cloud provisioning schemes have to over-provision because of the weak scalability of IaaS or CaaS. Now that ML serving is eligible for the highly scalable FaaS, we can reduce over-provisioning by combining IaaS and FaaS. IaaS is used as the primary serving option, while FaaS can provide transient service while new IaaS instances are launching. Moreover, FaaS can potentially handle the short lasting demand surges (short spikes), so that the overhead of frequent provisioning and deprovisioning can be eliminated. Although FaaS is costly, we believe the cost reduction from less over-provisioning can justify its price.

With IaaS as the primary serving option, we shall determine how to choose from a bewildering array of instance families and sizes, which we answer in the following subsections.

3.2 IaaS: Can we use burstable instances?

IaaS providers typically categorize instances into families. Within a family, instances share the similar physical hardware but may have various sizes in terms of vCPUs, memory, and network bandwidth. For CPU instances, EC2 offers four main instance families: the general-purpose m-family, the compute-optimized c-family, the burstable t-family, and the memory-optimized r-family.

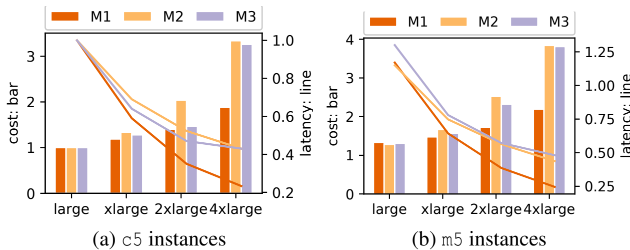
Among all instance types, burstable instances (t-family) have the lowest hourly rate, but they are aggressively multiplexed on overbooked servers [71, 73]. Burstable instances provide a baseline level (10% in AWS) of CPU performance with the ability to burst when required by the workload, yet with limited timespan according to a throttle policy (a new t2 instance can sustain 100% utilization for 30 minutes) [22, 23].

We profiled t2 instances' performance for ML serving and show the results in Table 3. We see that the latency drops linearly with the CPU allocation but adding more memory does

[2]Costs of instances are all based on AWS us-east-1 region.

Table 3: The average latency (t) and cost ($) of serving 1 million model inferences with bursted `t2` instances.

AWS `t2` Instance Size		micro	small	medium	large
Inception-v3	t (ms)	268.6	268.3	140.37	142.5
	$	0.87	1.71	1.81	3.75
Inception-ResNet	t (ms)	603.0	593.2	311.8	309.8
	$	1.94	3.79	4.01	7.96
OpenNMT-ende	t (s)	4.30	4.19	2.20	2.14
	$	13.85	24.83	28.36	56.71



(a) `c5` instances (b) `m5` instances

Figure 1: The latency (lines) and cost (bars) of serving 1 million model inference requests with `c5` and `m5` instances. M1, M2, and M3 respectively denote Inception-v3, Inception-ResNet, and OpenNMT-ende. The values are normalized by that of `c5.large` (182.5ms with $4.3 for M1; 389ms with $9.4 for M2; 2.18s with $51.5 for M3).

not benefit inference performance. Although it seems that `t2` instances are of low cost with viable latency for ML serving, these results are obtained in the bursted mode and do not sustain a long time. This fatal disadvantage means that burstable instances are not for compute-intensive services [50].

Summary. Burstable instances are plausible for transient ML serving usage, but not as the main long-running resources.

3.3 IaaS: Big instances or small instances?

We further investigate CPU instance families compute-optimized `c-family` and general-purpose `m-family`, where we focus on the latest generation `c5` and `m5`. We exclude memory-optimized instances (`r-family`) from consideration, as our measurements on `t2` instances indicate that 4GB of memory already does not bound the inference performance. In EC2, the configurations (vCPUs and memory) and prices of `m5` and `c5` instances are proportional to their sizes, so it is important to see how scaling up to larger instances would affect the ML serving performance.

Figs. 1a and 1b depict the measured latency (lines) and cost (bars) of serving 1 million inference requests of three ML models using `c5` and `m5` instances of different sizes. In general, `c5` instances are cheaper and have lower latency than `m5` instances because of more advanced CPU models, even though the latter have higher memory than the former. Our results also suggest that, for CPU instances of the same family, smaller instances are more cost-effective, as the serving throughput grows sub-linearly with the instance size. At the same time, by scaling from a smaller instance to a bigger one, the latency drops *sub-linearly* as well.

Summary. To sum up, smaller instances with advanced CPU models (`c5.large` in AWS) are preferable as they achieve higher performance-cost ratio. Moreover, owing to the finer provisioning granularity, using smaller instances to serve dynamic workload improves the resource utilization. Note that the cost analysis presented here is based in on-demand market. Once we switch to the spot market, the cost-effectiveness is variable w.r.t. the change of spot price.

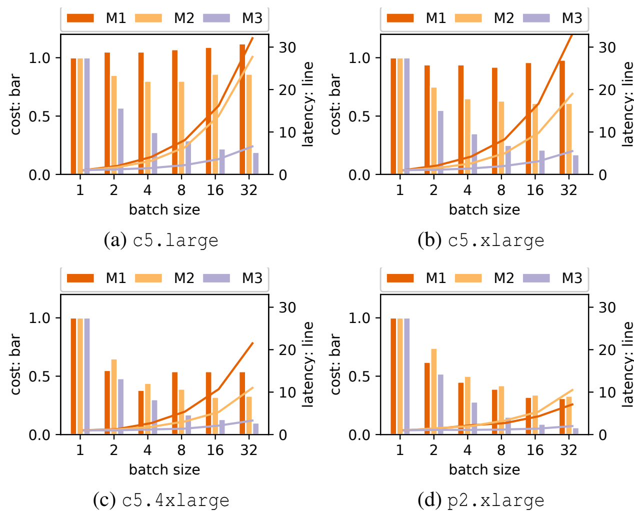
3.4 IaaS: How does GPU compare with CPU?

Many high-end IaaS instances are equipped with hardware accelerators, such as GPU and TPU (exclusive in Google Cloud), that can be used to speed up ML training and inference. The questions are: how would those hardware accelerators improve the latency of ML serving, and if such performance benefit can justify their high cost? In this subsection, we focus on GPU instances, as GPU is the most accessible and popular general-purpose ML accelerator. We will extend our study to TPUs in Google Cloud in §3.5.

A GPU instance is more expensive than a CPU instance, but it can achieve up to 40× speedup due to its massive parallel nature according to NVIDIA [54]. In order to unleash the full power of its computing capability, it is essential to *batch* multiple inference requests and serve them in one go [68]. Batching benefits the performance in two ways. First, it amortizes the overhead of operations such as RPC calls and inter-device memory copy. Second, it can take advantage of batch operation optimization from both software and hardware [30, 62].

To disclose the intriguing performance difference between CPU instances and GPU instances as well as batching, we compare the inference performance of three ML models on `c5` CPU instances and GPU instances `p2.xlarge`. We choose `p2.xlarge` as it is the smallest GPU instance in AWS (the next size available is `p2.8xlarge` which has 8 GPUs and is too expensive). Fig. 2 shows the cost and latency of serving 1 million inference requests with various batch sizes (# of requests served in one batch) on `c5` and `p2.xlarge` instances. For smaller CPU instances such as `c5.large` and `c5.xlarge`, the serving cost (bars) and latency improvement (lines) over batching is marginal (latency growing proportionally as the batch size), while bigger CPU instance (`c5.4xlarge`) displays certain improvement when batch size increases within a small range. GPU instances, on the other hand, benefit significantly from batching: the larger the batch, the lower the cost per request. This phenomenon suggests that batching can significantly improve the cost-effectiveness of larger CPU instances and GPU instances.

Summary. With an appropriate batch size, GPU instances can achieve lower per-request cost and shorter inference latency than CPU instances. However, batch size cannot be in-



(a) c5.large (b) c5.xlarge

(c) c5.4xlarge (d) p2.xlarge

Figure 2: The cost and batch latency of 1 million model inference with batching of various sizes. M1, M2, M3 represents inception-v3, inception-resnet, and OpenNMT-ende. The cost and batch latency are normalized by the values when batch size is set to 1.

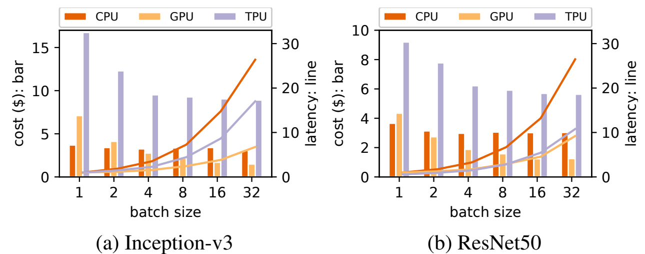
creased arbitrarily: increasing batch size leads to both longer queuing latency and batch inference latency [30]. We will further discuss the batching configuration in §4 and formulate the problem in a latency-aware context.

3.5 Characterization in Google Cloud

So far, all our profiling experiments are based on AWS. To validate whether our main observations also apply to ML serving in the other cloud platforms, we extend our characterization to Google Cloud [37] which offers similar service and pricing options as AWS, along with the Tensor Processing Unit (TPU), the state-of-the-art ML ASIC.

IaaS remains the best option. We first compare the cost and latency performance of ML serving using Google's IaaS, CaaS, and FaaS with the same workloads as in §3.1. All the experiments were run in `us-central1` region. Among the three provisioning options, IaaS remains the best with the lowest cost and shortest latency. For instance, the average latency and total cost of serving 1 million Inception-v3 requests on an customized IaaS instance with 1 vCPU and 2GB memory are 317ms and $3.70, respectively. In comparison, it takes 319ms and $4.17 using the cheapest CaaS instance `n1-standard-1` (1 vCPU and 3.75GB memory), and 527ms and $17.4 using Google Cloud Functions (FaaS) with 2GB memory.

Small instances win on performance-cost ratio. We then compare the cost and latency performance of CPU instances of various sizes within the same family. We made the similar observations as in AWS (§3.3): smaller instances offer higher performance-cost ratio than the bigger ones, though the latter leads to shorter latency. In particular, when serving 1 million Inception-v3 requests with `n1-standard-1`, `n1-standard-2`, and `n1-standard-4`, the cost (average latency) ends up with $4.16 (319ms), $7.82 (296ms), and $11.98 (227ms), respectively.



(a) Inception-v3 (b) ResNet50

Figure 3: The cost and batch latency of serving 1 million inference requests with various batch sizes. The batch latencies are normalized by the latency when there is no batching.

CPU, GPU, or TPU? Finally, we compare the cost and latency performance of using CPU, GPU, and TPU instances for ML serving with various batch sizes. We chose two popular image classification models, Inception-v3 and ResNet50 [44]. The results are shown in Fig. 3, where we used a customized CPU instance with 1 vCPU and 2 GB memory (`CPU`), the same instance with a K80 GPU attached to it (`GPU`), and a Cloud TPU-v2 instance (`TPU`). We observe the similar trend of cost and latency w.r.t. batch size for CPU and GPU instances as in AWS (§3.4). As for TPU, we find that its high price tag does not justify the performance benefit. In fact, TPU is a massively parallel accelerator optimized for training throughput rather than inference latency. Note that in Fig. 3, the batch size for TPU is calculated per core. As TPUv2 has 8 cores, the device batch size is actually 8 times the value. The design of TPU calls for large batch sizes to fully exploit its computing capacity [40]. However, the stringent latency requirement of real-time inference cannot wait for large batches to accumulate, leading to extremely low hardware utilization. In summary, TPUs are not suitable for real-time ML serving.

3.6 Characterization Summary

We summarize our key findings as follows: (1) IaaS achieves the best cost and latency performance for ML model serving, and combining it with FaaS can potentially reduce over-provisioning while remaining scalable to spiky workloads. (2) Burstable instances are viable to cover transient ML serving demand. (3) In on-demand CPU market, smaller instances have higher performance-cost ratio than the bigger ones, even though the latter provides shorter latency. (4) Only with appropriate batching can the use of GPU instances be justifiable to achieve lower cost and shorter latency than CPU instances.

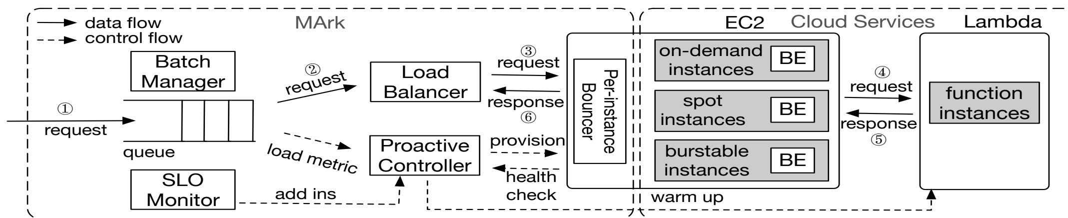


Figure 4: An overview of the MArk model serving system.

4 MArk

In this section, we present MArk (Model Ark), a scalable system that provides cost-effective, SLO-aware ML inference serving in AWS. While MArk is built in AWS, nothing prevents our design from being extended to the other cloud platforms with similar service offerings, such as Google Cloud.

4.1 Overview

Following our observations in §3, MArk uses EC2 as the primary means of provisioning ML serving. It also uses Lambda to quickly cover the service gap when there is a need to scale out/up. Fig. 4 illustrates the overall architecture of MArk. In particular, requests from clients are deposited to a request queue, and are grouped into batches by the *Batch Manager* (details in §4.3). MArk periodically measures the workload metrics, such as the request arrival rate, and sends them to a *Proactive Controller* which makes predictions and plans instances in advance to reduce over-provisioning (§2.2). The controller then sends the launching and destroying requests to EC2 instances, on which custom service backends such as Tensorflow Serving [55] are hosted. The controller also monitors the health status of all running instances. With predictive scaling, further actions are needed to handle prediction errors and unexpected load surges. On each running EC2 instance, there is a *Bouncer* monitoring serving metrics and performing request admission control. If an incoming request cannot be served within a specified time $RT_{\max}$, it will be handled by Lambda instances immediately. In addition, MArk employs an *SLO Monitor* that keeps track of and maintains the SLO compliance with the method described in §4.4.

SLO requirements. Following Swayam [41], we set two SLO requirements for MArk. (1) *Response Time Threshold*: A request is deemed fulfilled only if its response time is below $RT_{\max}$. (2) *Service Level*: The service is considered satisfactory only if at least $SL_{\min}$ percent of requests are fulfilled.

4.2 Workload Prediction

MArk employs predictive scaling to reduce over-provisioning. To expose the long-term cost trade-off between different instances and resource provisioning, we need to estimate the maximum request rate in the near future, which requires multi-step workload prediction. Existing works employ many well-established resource estimation methods, such as linear regression [27], autoregressive models [34, 61], and neural networks [19, 53, 57, 65]. As the accuracy of prediction depends on the underlying workload, there is no such a universal method that works perfectly in all cases. Therefore, MArk exposes an API through which users can implement their own workload prediction methods that best fit their applications. The challenge is how to gracefully handle unavoidable prediction errors and unexpected load surges.

We have implemented a vanilla version of long short-term memory (LSTM) network [36] for multi-step workload prediction, as it is reported to give the state-of-the-art performance [64]. In our implementation, the prediction unit (time interval) is P_u, and the prediction window is P_w, meaning MArk updates the predicted load for the next P_wP_u interval every P_u time units. During each unit, MArk keeps sampling the arrival rate in consecutive short sample windows of P_s. It keeps track of the maximum arrival rate of the unit, and gets the maximum arrival rate array for the next P_w units. In our evaluations, we set the $[P_u, P_w, P_s]$ to $[1\text{min}, 60, 5\text{s}]$. Prediction unit is set to 1 minute, as EC2 charges at least 1 minute for new instances. Prediction window is set to 60 steps, since 1 hour of future trend is good enough to expose the long term trade-offs. The sample size is set to 5 seconds, since the arrival rate can be treated as stable in short time slots [74]. MArk is designed to work for all ML serving workloads, so users can fine-tune this prediction algorithm or replace it with their own implementations for better prediction results.

4.3 Instance Provisioning and Batching

With workload prediction, we need to determine what and how many instances should be used to serve the requests. In general, this problem can be formulated as a compilation of queueing system [74], where instances of each type are modeled as an M/D/c queue with deterministic processing time and the predicted request arrival rate. However, as shown in [74], this problem has no closed-form solution even without considering request batching and instance pricing. Given this hardness result, we turn to a heuristic solution: instead of jointly considering batching and instance provisioning, we solve the two problems separately using heuristics.

Batching. Inspired by the adaptive batching in [30], we introduce two hyperparameters to control the batching behavior of an instance type: W_{batch} which is the maximum waiting

time window for request batching, and N_{batch} which is the maximum batch size. The Batch Manager fetches requests from the queue, and submits the batched requests if either of the two limits is reached (Fig. 4). We tune the two hyperparameters to meet the following two requirements: (1) No SLO requirements can be violated, meaning the waiting time window and the processing time of the batch together should be capped by response time threshold $RT_{\max}$; (2) the throughput with batching enabled must be greater than that of no batching. That is, the waiting time window and the batch processing time together should be less than the time needed to process all those requests sequentially without batching.

In practice, hyperparameter tuning requires light profiling for the target instance. We first profile the optimal processing rate of the target instance without batching, denoted by μ^*_{nb}. We then gradually increase the batch size from 1 until at least one of the following constraints no longer holds, where b is the batch size, and T_b is the time needed to process a batch:

$$W_{\text{batch}} + T_b \leq RT_{\max},$$
$$W_{\text{batch}} + T_b \leq \frac{b}{\mu^*_{nb}}.$$

Now that we have the optimal batch size $N_{batch} \leftarrow b$ and the maximum processing rate μ^* under this configuration, together with their corresponding W_{batch}, we can simply treat the target instance as a black box with processing rate μ^*.

Instance provisioning. We now solve the instance provisioning problem using an online heuristic algorithm that considers both long-term cost-effectiveness and the launch overhead, while at the same time attaining high utilization of running instances.

We first introduce the notations. Suppose there are n types of instances that can be used for serving. At a given time t_0, let $R = \{r_1, r_2, \cdots, r_n\}$ be the set of running instances and $F = (F_1, \cdots, F_m)$ the predicted maximum request arrival rate for the next m steps, where F_t is the predicted maximum rate in step t. For each instance type i, let C_i be the instance capacity, measured by the maximum throughput of a given model (requests per hour). Let P_i be its unit price, and O_i its launch overhead, i.e., cost due to the instance provisioning latency. Finally, let I be the set of available instance types. Given R, F, I and the target SLO, our problem is to determine what instances to launch and which instances to destroy at t_0, so as to minimize the cost while meeting the target SLO.

The challenge of finding the optimal solution in the long run is how to deal with the running instances at t_0. They may not be the most cost-effective in the next m steps, yet keeping using them avoids additional launch overhead. We propose a greedy solution in Algorithm 1. Our intuition is to greedily find the most cost-effective instance from time period t_0 to t_m considering both the pay-as-you-go fee and launch overhead. The running instances at t_0 can be treated as special ones without launch overhead.

Algorithm 1 Greedy Algorithm

```
procedure SCHEDULE(F, R, I, SLO)
    S ← S ∪ R    ▷ Running instances are treated as special ones with zero launch overhead
    for all instance i in S do
        if instance i cannot meet SLO requirement then
            S = S \ {i}                          ▷ Remove i from S
    if S = ∅ then
        Report error       ▷ No candidate instance can meet SLO
    instance_plan ← ∅               ▷ initialize provisioning plan
    FILL(F, S, instance_plan)
    Launch instances in instance_plan but not in R
    Destroy instances in R but not in instance_plan

procedure FILL(F, S, instance_plan)
    C^sum ← total capacity of all instance i in instance_plan
    for t = 1 to m do
        Λ_t = F_t − C^sum   ▷ Unfulfilled requests predicted at step t
    if Λ_τ ≤ 0 then          ▷ Planned capacity is enough at step τ
        return
    Find the largest e such that there are unfulfilled requests from steps τ to e, i.e., Λ_t ≤ 0 for all τ ≤ t ≤ e
    min_cost ← ∞ ▷ Greedily search the instance with the lowest per-request cost to cover unfilled requests from τ to e
    for all instance type i ∈ S do
        cost ← (O_i + (e − τ)P_i)/N, where N is the number of unfulfilled requests that will be served by an instance i in [τ, e]
        if cost < min_cost then
            min_cost ← cost
            j ← i
    instance_plan ← instance_plan ∪ {j}
    FILL(F, S, instance_plan)
```

In our algorithm, assuming most instances can get ready in τ time units after launching, we use the predicted load at $t_0 + \tau$ as the provisioning target, as it is safe to make instance provisioning decisions τ time units in advance. The values of τ can be easily adjusted based on the actual scenario. In our setup, τ is set to 5 minutes, and the scheduling time unit is set to 1 minute. In this case, the scheduling decisions are made every minute, targeting the load in 5 minutes. The launching requests should be sent right away once the *instance_plan* is ready, while destroying requests should be sent after a predefined cool-down period to ensure better service quality [59].

It is worth mentioning that Algorithm 1 trivially meets the SLO requirement by ensuring that the latency performance of each selected instance comply to the target SLO individually.

4.4 SLO tracking

The heuristic in Algorithm 1 plans instance capacity based on predictions. Yet not all demand surges are predictable, and such surges would result in SLO violations if solely relying on proactive provisioning [59]. To further improve the SLO compliance, MArk actively monitors request latency,

and *reactively scales* the cluster as soon as SLO violations are detected. MArk constantly checks if the last M requests satisfy the SLO requirements, if not, L instances of type T will be launched (`c5.large` by default). All those parameters can be tuned for specific models and SLO requirements.

4.5 Spot Instance and Lambda Cold Start

Use of spot instances. Note that Algorithm 1 does not differentiate between on-demand and spot instances, which allows MArk to exploit the price discount of spot instances. However, the adoption of spot instances poses the challenge of instance interruptions. Although the interruption of a spot instance will be notified 2 minutes in advance, such a grace period may not be long enough for a substitute spot instance to get ready. The question is how can we handle the outstanding requests in the presence of instance interruptions? Lambda seems to be a choice, but it would take a toll on the latency and cost.

Our answer to this challenge is the burstable instance. As shown in §3.2, burstable instances are cheap instances which can sustain full utilization for about 30 minutes. The low cost and high peak performance make them a perfect fit for transient backups in case of short-term interruptions. Moreover, burstable instances can be resumed from stopped state in less than 2 minutes thanks to their small sizes. Therefore, when we use spot instances with MArk, we reserve a few stopped burstable instances as cold standbys. Once MArk receives interruption notices, it resumes the corresponding amount of burstable instances to handle the transient requests until the regular spot instances capacity is back to normal, after which those burstable instances are stopped.

Lambda cold start. Another potential challenge MArk faces is the *cold starts* in Lambda [71]. Every time a new Lambda instance is launched, it needs to load the ML model, framework library and code in memory, which results in a much longer inference delay. Nevertheless, cold starts only occur when the request rate exceeds the concurrency, measured by the number of currently available lambda instances [32,73]. Existing benchmarking shows that a Lambda instance is recycled after it stays inactive for 45 to 60 minutes [31]. Our evaluations further confirm that, with more than 3 million requests, the cold start rate never exceeds 0.23%. Therefore, the latency impact of cold starts is limited. The cost impact is also negligible. Our profiling shows that $1 can spin up 7K inception-v3 Lambda instances, which is capable of serving more than 20K requests per second. Algorithm 1 hence does not consider the cost impact of Lambda cold starts.

Despite the negligible impacts of Lambda cold start, our implementation employs strategical concurrency warm-up to further amortize its impact. When a potential Lambda request surge is expected, such as spot interruptions and unexpected workload surges, MArk sends concurrent pings to Lambda to warm up more instances as described in [32].

Table 4: ML models and frameworks used in evaluation.

Model	Type	Framework	Size
Inception-v3	Image Classification	Tensorflow Serving	45MB
NASNet	Image Classification	Keras	343MB
LSTM-ptb	Language Modeling	MXNet Model Server	16MB
OpenNMT-ende	Machine Translation	Tensorflow Serving	330MB

5 Experimental Evaluation

We have prototyped the proposed MArk system and conducted extensive experimental evaluations on AWS to validate its effectiveness and robustness. We first compare the performance of MArk using on-demand instances and spot instances respectively with the premier industrial ML platform SageMaker against production traces from Twitter. To ensure MArk's performance does not mainly rely on prediction accuracy, we then examine whether MArk is able to maintain its advantage under unpredictable, highly bursty workload. After that, we run a few microbenchmarks to demonstrate the robustness of MArk in terms of handling spot interruptions, and the ability to handle unexpected demand surges.

5.1 Evaluation Setup

MArk. We have prototyped MArk on top of Amazon EC2 and Lambda services in two versions, *MArk-ondemand* which only uses on-demand instances, and *MArk-spot* which uses spot instances with interruption-tolerant mechanism, i.e., using burstable servers for smooth transition during unexpected instance interruption (§4.5).

Testbed. We use AWS as the testbed for conducting extensive experiments. The types of instance used in our evaluation include all the `c5` and `m5` instances as examples of CPU instances and `p2.xlarge` instances as an example of GPU accelerators. In our experiments, we used up to 42 `c5` instances, 10 `m5` instances, and 12 `p2.xlarge` instances.

ML models. We use four popular ML models that are of various sizes and cover diverse domains deployed in three popular ML serving software frameworks to evaluate MArk's performance, which are summarized in Table 4. To configure the batching of the ML models on EC2 instance, we performed lightweight profiling following the instructions detailed in §4.3. The optimal batching hyperparameters W_{batch} and N_{batch} for `p2.xlarge` instance found by our tuning algorithm outlined in §4.3 are 200ms and 8 for Inception-v3, 750ms and 16 for NASNet, 490ms and 16 for OpenNMT-ende. For LSTM-ptb, we only performed experiments on CPU as MXNet Model Server does not support batching at the time of writing. For OpenNMT-ende on CPU instance, the optimal batching hyperparameter N_{batch} is found to be 2, and W_{batch} is set accordingly. For the other models on CPU instance, we do not use batching as it does not bring benefits (see Fig. 2).

SLO. Recall that the SLO requirement is specified as at

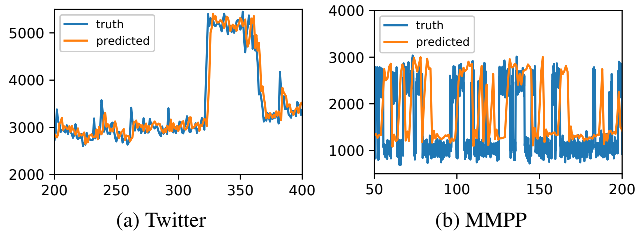


Figure 5: Snapshots of the arrival process using Twitter and MMPP with the prediction results of LSTM based algorithm.

least $SL_{\min}$ percent of requests must be served in $RT_{\max}$ time (§4.1). We set $SL_{\min}$ to 98% for all models, and set $RT_{\max}$ as 600ms, 1000ms, 100ms, and 1400ms for Inception-v3, NASNet, LSTM-ptb, and OpenNMT-ende respectively.

Workload. In our evaluation, we drive the arrival process of ML workloads in two different ways. First, as there is no publicly available traces for ML serving, we synthesize ML requests based on the tweets traces from Twitter [20]. We believe that the Twitter traces serve as a good benchmark, as it represents a popular web service with highly dynamic load. The trace exhibits typically characteristics of ML inference workloads, containing recurring patterns (e.g., hour of the day, day of the week) as well as unpredictable load spikes (e.g., breaking news). In particular, the peak request rate in the traces is 4 times higher than the valley, a result of transient demand surges commonly found in industrial-scale web applications. Fig. 5a(a) illustrates a snapshot of the trace.

Second, to further evaluate the performance sensitivity of MArk w.r.t the workload, we synthesize random and bursty ML request load using *Markov-Modulated Poisson process* (MMPP) [28,35,60]. The load generated by MMPP are highly unpredictable, as the occurrence and duration of demand surges are completely random, as shown in Fig. 5b.

In summary, we use the Twitter traces to evaluate how well MArk performs against synthesized real workload that can be largely predicted. Using MMPP-generated workload, we stress test MArk's performance in the presence of frequent, unpredictable load spikes.

Baseline. We use SageMaker [13] as the baseline for the evaluation. SageMaker is AWS's leading ML training and hosting system. SageMaker hosting employs AWS's new target tracking autoscaling policy [16,18]. Given the dynamics in request arrival rate (i.e., the arrival rate can increase more than double in just a few minutes), to ensure service quality, we follow the AWS guidelines [16] and set the over-provisioning factor to 2 for SageMaker. We will show in Fig. 7 that even so the over-provisioning is still incapable of handling the volatile workload of the Twitter traces.

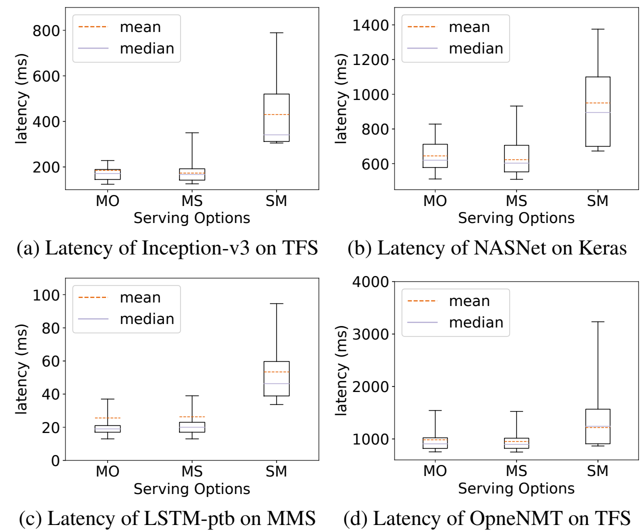


Figure 6: Latency comparison of MArk-ondemand (MO), MArk-spot (MS), and SageMaker (SM) on 4 ML models using Twitter workload.

5.2 Macrobenchmarks

Workload prediction. For Twitter traces, we use the data of the first 5 months to train the workload prediction model. For MMPP-generated arrival process, we use a period of 24-hour data for training. Fig. 5b demonstrates snapshots of the prediction results. We see that the prediction accuracy is in general good for the Twitter traces, yet very poor for the MMPP case. Since striving for the best workload prediction is NOT the focus of this paper, and we mainly use the LSTM based algorithm as an example of the pluggable workload prediction component, we do not provide detailed evaluation of the prediction algorithm in the interest of space.

Experimental results using Twitter traces. We first compare MArk-ondemand, MArk-spot, and SageMaker on the ML models described in §5.1 by feeding the arrival rate extracted from Twitter traces. The experiments were performed on AWS spanning more than 8 hours each. We report two metrics: request latency in Fig. 6, and cost breakdown in Table 5. The request latency is measured as the time between request arriving at the serving system and getting response back, while the cost is the charge billed by AWS. The comparison results suggest that MArk can significantly reduce both the cost and latency compared with SageMaker. For cost reduction, compared with SageMaker, MArk-ondemand respectively achieves 3.63×, 2.79×, 2.41×, and 3.15× for the four ML models; MArk-spot achieves 6.21×, 5.91×, 6.64×, and 7.83×, respectively. For latency, MArk-ondemand achieves up to 57% reduction and MArk-spot achieves up to 60% reduction compared with SageMaker.

The latency advantage of MArk over SageMaker comes in three-fold. First, with appropriate batching configuration,

Table 5: Cost ($) comparison of MArk-ondemand (MO), MArk-spot (MS), and SageMaker (SM) on 4 ML models using Twitter workload.

Setting	Inception-v3			NASNet		
	MO	MS	SM	MO	MS	SM
EC2	20.94	9.83	80.98	24.21	10.71	68.1
Lambda	1.34	3.2	NA	0.19	0.81	NA
Total	22.28	13.03	80.98	24.40	11.52	68.1

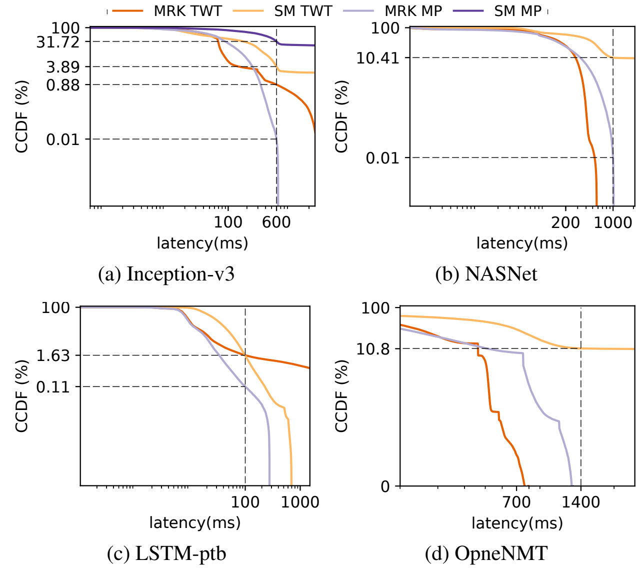
Setting	LSTM-ptb			OpenNMT-ende		
	MO	MS	SM	MO	MS	SM
EC2	6.17	2.24	14.9	27.54	10.79	87.1
Lambda	0	0.04	NA	0.12	0.33	NA
Total	6.17	2.28	14.9	27.66	11.12	87.1

GPU instances can reduce the overall latency by performing more efficient parallel computation. Second, the SLO-aware design of MArk helps reduce the queuing delay. In addition, the predictive scaling and SLO-awareness together form an efficient hybrid approach that enjoys the benefits in both proactive and reactive designs. It is worth pointing out the different performance behaviors between MArk-ondemand and MArk-spot. As shown in the latency box plots in Fig. 6, MArk-spot has longer latency tails, since more requests are handled by Lambda compared with MArk-ondemand, in case of interruptions. However, the average and median latencies of MArk-spot are usually the same or even better than MArk-ondemand. This is because in spot market, the performance-cost ratio is highly dynamic, which allows MArk-spot to opportunistically use large instances and GPU instances at cheaper price than on-demand, leading to better latency performance.

MArk's cost reduction comes from the following aspects. First, predictive scaling together with Lambda services brings a more judicious over-provisioning design that can reduce the cost. The 2× cost reduction over SageMaker in MArk-ondemand using only CPU instances for LSTM-ptb is a good example. Second, GPU instances can further reduce the cost during high arrival rate as batching increases the efficiency of computing. The cost reduction is more significant for OpenNMT as it benefits the most from batching as shown in Fig. 2d. MArk-spot further brings down the cost by enjoying the spot market discounts. Note that although Lambda service used by MArk is expensive in price, but the cost of Lambda can be well justified by enabling more judicious over-provisioning.

We have also performed a case study of SLO compliance and report the *Complementary Cumulative Distribution Function* (CCDF) of request latency in Fig. 7. As expected, MArk managed to maintain its compliance with SLO requirements, thanks to the SLO-aware design. SageMaker, on the other hand, is SLO-oblivious, so the queuing delay adds up during high arrival periods, and the SLO is violated.

Experimental results using MMPP-generated load. Next we evaluate MArk using the more challenging, less predictable MMPP workload. We still use the same four ML models, and each experiment lasts about 4 hours on AWS. In the interest of space, we only demonstrate the SLO compliance results in Fig. 7. Fig. 7a shows that the SLO compliance of SageMaker is significantly degraded from Twitter case to MMPP case due to the much more dynamic and bursty behaviors in MMPP. However, MArk can still meet the SLO requirements even when the workload is highly dynamic and unpredictable, thanks to the SLO Monitor that can detect the failure of proactive prediction and timely add backup machines based on the feedback control algorithm. Note that we only evaluated SageMaker with MMPP-driven arrival process on Inception-v3 model as it is too expensive for us to run all of them. However, given the SLO-oblivious nature of SageMaker, we expect the behavior would be similar.



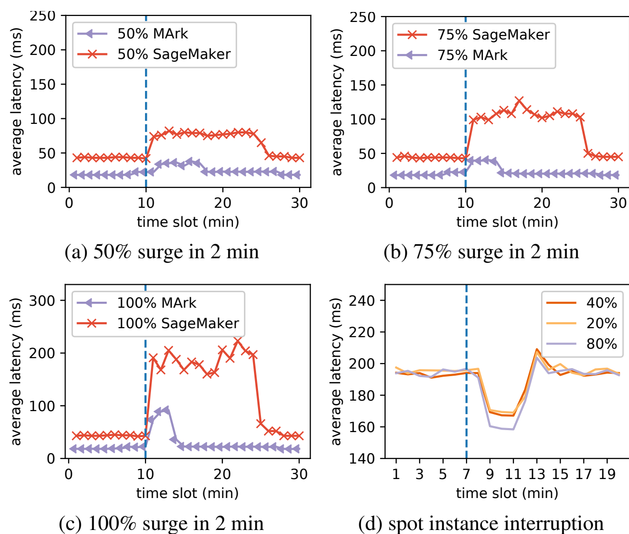
(a) Inception-v3 (b) NASNet (c) LSTM-ptb (d) OpneNMT

Figure 7: CCDF of latency comparison between MArk and SageMaker. RT_{max} is drawn as a black dashed vertical line (the black dashed horizontal line shows the corresponding CCDF value of RT_{max}). MRK and SM represents MArk and SageMaker, while TWT and MP represents Twitter and MMPP workload respectively.

5.3 Microbenchmarks

In this section, we evaluate the robustness of MArk by taking a closer look at how MArk handles unexpected demand surges and spot interruptions.

Robustness against unexpected surge. MArk harvests performance and cost benefits by using a judicious over-provisioning scheme. One important question is whether MArk can handle unexpected demand surges well in the presence of unforeseeable flash crowds or poor workload prediction accuracy. To answer this question, we increase the request rate for LSTM-ptb serving by 50%, 75%, and 100% in 2 minutes and compare the latency over time between MArk



(a) 50% surge in 2 min (b) 75% surge in 2 min (c) 100% surge in 2 min (d) spot instance interruption

Figure 8: Microbenchmark results. (a), (b), (c): The latency change comparison during unexpected demand surge between MArk and SageMaker, where the surge starts at the 11th min shown by the dashed line. (d): The latency change when different percentages of spot instances are interrupted in MArk-spot, where the interruption notice is received at the 7th min.

and SageMaker in Figs. 8a, 8b, and 8c. [3] Since the surge is unpredictable, both MArk and SageMaker handle it reactively. The results suggest that MArk acts faster and effectively than SageMaker during the unforeseeable surge, i.e., the increased latency period and amount are much smaller, thanks to the Lambda-based fallback mechanism, which can immediately take over and cap the latency to prevent queue building up like in SageMaker. In addition, MArk's SLO Monitor can detect the SLO violations and issue backup instance requests right away to adapt to the new arrival rate, while SageMaker is only able to react in the next scaling cycle.

Robustness against spot interruption. MArk-spot utilizes spot instances to reduce the cost. However, the interruption of spot instance can cause performance degradation if not handled properly. We evaluate MArk-spot by zooming in the interruption handling periods under different interruption ratio of instances. We launched a 20-instance Inception-v3 cluster, and manually interrupted 20%, 40%, and 80% of the instances respectively. Fig. 8d illustrates the latency change during the interruption. The interruption happens at the 7th minute (vertical dashed line), and MArk resumes `t2` instances as transient resources upon receiving interruption notice. The proactive controller then adjusts the provisioning plan and requests new instances. At the 13th minute new spot instances are ready, and the latency goes back to normal. The average latency drops during transient period because burstable `t2` instances can have temporal boosted performance as discussed in §3.2. The short latency bump at the 13th minute is due to the switching overhead (i.e., warm up of new instances).

To sum up, the results above confirm that MArk can handle unexpected surge and spot interruption robustly.

[3] Given that we only compare latency here, we show the results of MArk-spot as the latency results of MArk-ondemand can only be better.

6 Discussion

Cloud platform. The measurements and evaluations in this paper are mainly based on AWS. However, the main design of MArk can be generally extended to other major cloud platforms, as they offer both IaaS and FaaS services, as well as flexible pricing models. Nevertheless, some hyperparameters used in the algorithm are platform-dependent, and must be re-tuned. Also, we have not considered reserved instances, as they require a long-term usage commitment. We believe they will bring down the cost of serving stable inference demands in a long run, and will leave it as a future work.

Large models. Deep learning models are becoming increasingly large and may not fit into the memory of Lambda (or even IaaS) instances. A possible solution goes to distributed inference under the model parallel scheme, which is not supported in our current design. We will leave it as a future work.

Hardware accelerator. We used the most common ML accelerator GPU as an example of utilizing hardware accelerators. The same batching formulation can be applied to other accelerators (e.g., FPGA) as they benefit from batching similarly.

MArk's architecture requires a master machine to make provisioning decisions. While such design has limitations on scalability and is vulnerable to the single point of failure, these problems can be easily addressed with mature industrial solutions such as Zookeeper [46].

7 Concluding Remark

In this paper, we conducted a systematic study of serving ML models on cloud and concluded that combining FaaS and IaaS can achieve scalable ML serving with low over-provisioning cost. Driven by the unique characteristics of ML model serving, we proposed MArk, a cost-effective and SLO-aware ML serving system. We prototyped MArk on AWS and showed that compared with the premier autoscaling ML platform SageMaker, MArk yields significant cost reduction (up to 7.8×) while complying with the SLO requirements with even better latency performance.

Acknowledgement

This work was supported in part by RGC ECS grant 26213818, NSF grant CCF-1756013, and IIS-1838024 (using resources provided by AWS as part of the NSF BIGDATA program). Chengliang Zhang and Minchen Yu were supported by the Hong Kong PhD Fellowship Scheme and the Huawei PhD Fellowship Scheme, respectively.

References

[1] Amazon EC2. https://aws.amazon.com/ec2/, 2018.

[2] Amazon ECS. https://aws.amazon.com/ecs/, 2018.

[3] AWS Lambda. https://aws.amazon.com/lambda/, 2018.

[4] Docker. https://www.docker.com, 2018.

[5] Google Cloud Functions. https://cloud.google.com/functions/, 2018.

[6] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/, 2018.

[7] PredictionIO. https://predictionio.apache.org, 2018.

[8] RedisML. https://github.com/RedisLabsModules/redis-ml, 2018.

[9] ALI-ELDIN, A., KIHL, M., TORDSSON, J., AND ELMROTH, E. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd ACM Workshop on Scientific Cloud Computing* (2012).

[10] ALI-ELDIN, A., TORDSSON, J., AND ELMROTH, E. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE Network Operations and Management Symposium* (2012).

[11] AMAZON. Amazon Web Services. https://aws.amazon.com/, 2018.

[12] AMAZON. AWS autoscaling. https://aws.amazon.com/autoscaling/, 2018.

[13] AMAZON. Build, train, and deploy machine learning models at scale. https://aws.amazon.com/sagemaker/, 2018.

[14] AMAZON. Configuring Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, 2018.

[15] AMAZON. Dynamic scaling for Amazon EC2 auto scaling. https://amzn.to/2W2jvhc, 2018.

[16] AMAZON. Load testing for variant automatic scaling. https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html, 2018.

[17] AMAZON. New Amazon EC2 spot pricing model: Simplified purchasing without bidding and fewer interruptions. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/, 2018.

[18] AMAZON. Target tracking scaling policies for Amazon EC2 auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html, 2018.

[19] ANIELLO, L., BONOMI, S., LOMBARDI, F., ZELLI, A., AND BALDONI, R. An architecture for automatic scaling of replicated services. In *Networked Systems*. Springer, 2014, pp. 122–137.

[20] ARCHIVETEAM. Twitter streaming traces, 2017.

[21] AWS. Amazon EC2 reserved instances. https://aws.amazon.com/ec2/pricing/reserved-instances/, 2018.

[22] AWS. Burstable performance instances. https://amzn.to/2APg4hG, 2018.

[23] AWS. Right sizing: Provisioning instances to match workloads. https://amzn.to/2VdIiK9, 2018.

[24] AWSLABS. MXNet model server. https://github.com/awslabs/mxnet-model-server, 2018.

[25] BARRETT, E., HOWLEY, E., AND DUGGAN, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience 25*, 12 (2013), 1656–1674.

[26] BERGSTRA, J., BASTIEN, F., BREULEUX, O., LAMBLIN, P., PASCANU, R., DELALLEAU, O., DESJARDINS, G., WARDE-FARLEY, D., GOODFELLOW, I., BERGERON, A., ET AL. Theano: Deep learning on GPUs with Python. In *NeuralPS, Big Learning Workshop* (2011).

[27] BODÍK, P., GRIFFITH, R., SUTTON, C., FOX, A., JORDAN, M. I., AND PATTERSON, D. A. Statistical machine learning makes automatic control practical for internet datacenters. In *USENIX HotCloud* (2009).

[28] CASALE, G., ZHANG, E. Z., AND SMIRNI, E. Trace data characterization and fitting for markov modeling. *Perform. Eval. 67*, 2 (2010), 61–79.

[29] CHOLLET, F., ET AL. Keras: Deep learning library for Theano and TensorFlow. https://keras.io, 2015.

[30] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A low-latency online prediction serving system. In *NSDI* (2017), pp. 613–627.

[31] CUI, Y. How long does AWS Lambda keep your idle functions around before a cold start? https://bit.ly/2tb7bLJ, 2018.

[32] CUI, Y. I'm afraid you're thinking about aws lambda cold starts all wrong. https://bit.ly/2Q1rrcr, 2018.

[33] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., AND VAHDAT, A. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems* (2003), vol. 4, pp. 5–5.

[34] FANG, W., LU, Z., WU, J., AND CAO, Z. Rpps: a novel resource prediction and provisioning scheme in cloud data center. In *IEEE International Conference on Services Computing* (2012).

[35] FISCHER, W., AND MEIER-HELLSTERN, K. The Markov-modulated Poisson process (MMPP) cookbook. *Perform. Eval. 18*, 2 (1993), 149–171.

[36] GERS, F. A., SCHMIDHUBER, J., AND CUMMINS, F. Learning to forget: Continual prediction with LSTM. In *9th International Conference on Artificial Neural Networks* (1999).

[37] GOOGLE. Google cloud. https://cloud.google.com/, 2018.

[38] GOOGLE. Google cloud autoscaling. https://cloud.google.com/compute/docs/autoscaler/, 2018.

[39] GOOGLE. Kubernetes horizontal scaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2018.

[40] GOOGLE. Cloud TPU performance guide. https://cloud.google.com/tpu/docs/performance-guide, 2019.

[41] GUJARATI, A., ELNIKETY, S., HE, Y., MCKINLEY, K. S., AND BRANDENBURG, B. B. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 109–120.

[42] HAN, R., GHANEM, M. M., GUO, L., GUO, Y., AND OSMOND, M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems 32* (2014), 82–98.

[43] HARLAP, A., TUMANOV, A., CHUNG, A., GANGER, G. R., AND GIBBONS, P. B. Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets. In *Proceedings of ACM EuroSys* (2017).

[44] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of IEEE CVPR* (2016).

[45] HE, X., SHENOY, P., SITARAMAN, R., AND IRWIN, D. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), ACM, pp. 207–218.

[46] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX ATC* (2010).

[47] KLEIN, G., KIM, Y., DENG, Y., SENELLART, J., AND RUSH, A. M. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[48] Lee, H., Satyam, K., and Fox, G. Evaluation of production serverless computing environments. In *Proceedings of IEEE CLOUD* (2018).

[49] Lee, Y., Scolari, A., Chun, B.-G., Santambrogio, M. D., Weimer, M., and Interlandi, M. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proceedings of USENIX OSDI* (2018).

[50] Leitner, P., and Scheuner, J. Bursting with possibilities: An empirical study of credit-based bursting cloud instance types. In *Proceedings of IEEE/ACM Utility and Cloud Computing* (2015).

[51] Merity, S., Keskar, N. S., and Socher, R. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[52] Microsoft. Microsoft Azure cloud computing platform & services. https://azure.microsoft.com/en-us/, 2018.

[53] Nikravesh, A. Y., Ajila, S. A., and Lung, C.-H. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Proceedings of IEEE International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2015).

[54] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2018.

[55] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. TensorFlow-Serving: Flexible, high-performance ML serving. *arXiv preprint arXiv:1712.06139* (2017).

[56] Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of ACM EuroSys* (2018).

[57] Prodan, R., and Nae, V. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems 25*, 7 (2009), 785–793.

[58] Qu, C., Calheiros, R. N., and Buyya, R. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications 65* (2016), 167–180.

[59] Qu, C., Calheiros, R. N., and Buyya, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 73.

[60] Rajabi, A., and Wong, J. W. MMPP characterization of web application traffic. In *Proceedings of IEEE MASCOTS* (2012).

[61] Roy, N., Dubey, A., and Gokhale, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proceedings of IEEE CLOUD* (2011).

[62] Sanders, J., and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[63] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of ACM EuroSys* (2015).

[64] Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Proc. NeuralPS* (2015).

[65] Song, B., Yu, Y., Zhou, Y., Wang, Z., and Du, S. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing* (2017), 1–15.

[66] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2017), vol. 4, p. 12.

[67] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE CVPR* (2016).

[68] TensorFlow. TensorFlow Serving batching guide. https://bit.ly/2VOpb9O, 2018.

[69] Tu, Z., Li, M., and Lin, J. Pay-per-request deployment of neural network models using serverless architectures. In *Proceedings of NAACL: Demonstrations* (2018).

[70] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3*, 1 (2008), 1.

[71] Wang, C., Urgaonkar, B., Gupta, A., Kesidis, G., and Liang, Q. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of ACM EuroSys* (2017).

[72] Wang, W., Wang, S., Gao, J., Zhang, M., Chen, G., Ng, T. K., and Ooi, B. C. Rafiki: Machine learning as an analytics service system. *arXiv preprint arXiv:1804.06087* (2018).

[73] Yan, F., Ren, L., Dubois, D. J., Casale, G., Wen, J., and Smirni, E. How to supercharge the amazon t2: Observations and suggestions. In *Proceedings of IEEE CLOUD* (2017).

[74] Yan, F., Ruwase, O., He, Y., and Smirni, E. SERF: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of IEEE/ACM SC16* (2016).

[75] Zhang, H., Stafman, L., Or, A., and Freedman, M. J. SLAQ: Quality-driven scheduling for distributed machine learning. In *Proceedings of ACM SoCC* (2017).

[76] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012 2*, 6 (2017).

Cross-dataset Time Series Anomaly Detection for Cloud Systems

Xu Zhang[1,2], Qingwei Lin[2], Yong Xu[2], Si Qin[2], Hongyu Zhang[3], Bo Qiao[2], Yingnong Dang[4], Xinsheng Yang[4], Qian Cheng[4], Murali Chintalapati[4], Youjiang Wu[4], Ken Hsieh[4], Kaixin Sui[2], Xin Meng[2], Yaohai Xu[2], Wenchi Zhang[2], Furao Shen[1], and Dongmei Zhang[2]

[1]*Nanjing University, Nanjing, China*
[2]*Microsoft Research, Beijing, China*
[3]*The University of Newcastle, NSW, Australia*
[4]*Microsoft Azure, Redmond, USA*

Abstract

In recent years, software applications are increasingly deployed as online services on cloud computing platforms. It is important to detect anomalies in cloud systems in order to maintain high service availability. However, given the velocity, volume, and diversified nature of cloud monitoring data, it is difficult to obtain sufficient labelled data to build an accurate anomaly detection model. In this paper, we propose cross-dataset anomaly detection: detect anomalies in a new unlabelled dataset (the target) by training an anomaly detection model on existing labelled datasets (the source). Our approach, called ATAD (Active Transfer Anomaly Detection), integrates both transfer learning and active learning techniques. Transfer learning is applied to transfer knowledge from the source dataset to the target dataset, and active learning is applied to determine informative labels of a small part of samples from unlabelled datasets. Through experiments, we show that ATAD is effective in cross-dataset time series anomaly detection. Furthermore, we only need to label about 1%-5% of unlabelled data and can still achieve significant performance improvement.

1 Introduction

In recent years, we have witnessed increasing adoption of cloud service systems. Many software applications are now deployed on cloud computing platforms such as Microsoft Azure, Google Cloud Platform, and Amazon Web Services (AWS). As the cloud systems could be used by millions of users around the world on a 24/7 basis, high service reliability and availability are critical.

However, cloud systems, like other software systems, may exhibit some anomalous behaviors. These anomalies could seriously affect service availability and reliability and could even lead to huge financial loss. The anomalies could be caused by a variety of factors (such as software bugs, disk failures, memory leaks, network outage, etc.) and reflected by a variety of cloud monitoring data (such as KPI, performance counters, usage statistics, system metrics, logs, etc.). The cloud monitoring data are usually time series data, which have high velocity and enormous volume because of the scale and complexity of cloud systems. To maintain high service reliability and availability, it is important yet challenging to detect anomalies from a large amount of cloud monitoring data precisely and timely.

Specifically, anomaly detection in practice encounters several challenges due to the characteristics of cloud systems. A large cloud system is composed of a variety of services and each service is associated with some monitoring data. For some types of data, the characteristics of anomalies are common across many services. While for some other types of data, the characteristics of anomalies could differ from service to service. For example, 90% CPU utilization is normal for computation intensive services but anomalous for other services. Therefore, simple threshold-based anomaly detectors can hardly perform well for a variety of services.

Over the years, many machine-learning based anomaly detection methods have been proposed, including supervised [22, 25] and unsupervised methods [2, 41, 38]. However, it is not trivial to detect anomalies in a large and diversified set of time series data in real cloud environment where labelled data is scarce but a high detection performance is demanded. Unsupervised learning methods can deal with a large amount of data as they do not require labelled data. However, the performance achieved by these methods is rather low [13]. Although supervised learning methods can achieve higher accuracy than the unsupervised counterparts, it is time-consuming and tedious to manually label the anomalies due to the volume and diversity of cloud monitoring data. Therefore, supervised-learning based methods are difficult to be applied to anomaly detection in practice.

Facing the above challenges, to build an accurate and efficient anomaly detection model, we propose ATAD, which enables cross-dataset anomaly detection for cloud systems. The main idea of cross-dataset is to perform anomaly detection on an unlabelled dataset (the target dataset) by learning

from existing, labelled datasets (the source datasets). For example, a detector can be learned from a public dataset such as NAB [23], and then applied to an unlabelled dataset collected from a real-world system.

ATAD consists of two major components: 1) Transfer Learning, which transfers the common anomalous behavior learned from a labelled time series data to a large volume of target unlabelled dataset. Through transfer learning, the commonalities across datasets could be leveraged and the labelling effort for the target dataset could be reduced. 2) Active Learning, which improves the detection performance by labelling only a small number of selected samples in the target dataset. Through active learning, the diversified data with specific characteristics can be addressed with a small amount of labelling effort.

In particular, in the Transfer Learning component, we identify multiple features of cloud monitoring data and perform clustering to select an appropriate subset of existing labelled data as the sub source domain. Then the CORAL algorithm [37] is applied to narrow the feature difference between the source and target domain. In the Active Learning component, we utilize the UCD (Uncertainty-Context-Diversity) method to recommend informative data points to be labelled. The labelled points are used to retrain the classifier trained from the Transfer Learning component. In this way, we aim at minimizing the labelling effort and improving the performance of the detector as much as possible.

We have conducted experiments on public datasets to verify the effectiveness of our method. The experimental results show that using ATAD we can achieve cross-dataset anomaly detection with good accuracy. We test the effectiveness on both public datasets and real-world cloud monitoring data. On public datasets, ATAD shows higher accuracy than existing methods when performing anomaly detection on a target dataset (i.e. Yahoo) by the detector learned from a source dataset (Non-Yahoo, like AWS, Twitter and Artificial datasets). Furthermore, labelling about 1%-5% of unlabelled data could achieve much higher F1-score than the related methods. We also train an ATAD model using public datasets and apply the trained model to detect anomalies in real-world cloud monitoring data of Microsoft. ATAD achieves the best F1-Score, which is much higher than those achieved by other methods.

The contributions of this paper are as follows:

- We propose a new anomaly detection method called ATAD, which enables cross-dataset anomaly detection for cloud systems.
- To the best of our knowledge, we are among the first to detect anomalies in time series cloud data using a combination of transfer learning and active learning techniques.
- We have performed an extensive evaluation of the proposed approach using public and real-world datasets.

This paper is organized as follows: we first elaborate the background and motivation of our work in Section 2. In Section 3, the details of ATAD are described. Section 4 reports the experiments and corresponding results. Next, we discuss threats to validity in Section 5. We introduce the related work in Section 6, before concluding the paper in Section 7.

2 Background and Motivation

For cloud service vendors like Microsoft Azure, Google Cloud Platform, and Amazon Web Services (AWS), there are millions of servers and virtual machines providing a variety of services to users. Despite many quality assurance methods, it is difficult to avoid system failures in reality. A severe system failure can cause damage to user's operation and vendor's reputation. Recovering system from failures in time is of great importance. In order to do that, quick and accurate anomaly detection is essential.

Anomaly detection is the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data [43]. Anomaly detection in cloud is usually performed on Cloud Monitoring Data (such as KPI, performance counters, CPU utilization, VM downtime, system workload, etc.). The cloud monitoring data is often presented in time series, that is a series of numerical data points recorded in time order.

Unlike a general anomaly detection problem, it is much more difficult to detect anomalies in a large-scale cloud service system. We identify the following challenges:

- Diverse characteristics of anomalies: in a large-scale cloud service system, different usage scenarios and components have different levels of tolerance to anomalies. For example, a minor system deviation occurring in a certain key component, like storage cluster, may become an anomaly and lead to the failure of the whole system [11, 21]. However, such a deviation may not cause serious problems in other components. It is difficult to set accurate thresholds of anomalies for each usage scenario and system component [11]. Therefore simple threshold-based anomaly detection methods are not suitable for cloud service systems.

- Anomaly detection in time series data: cloud monitoring data is large-scale time series data that has temporal property. Many commonly-used machine learning algorithms cannot be directly applied because the time series data does not satisfy the independent and identically distributed (i.i.d) assumption. Although some deep learning models, like LSTM [17], could capture the temporal property, they require enormous labelled data to train an accurate model. Thus, an appropriate approach to incorporate the temporal property of time series data is important.

- Unsatisfactory performance of unsupervised learning: unsupervised machine learning techniques such as Isolation Forest [26] or Seasonal Hybrid ESD [16] can be applied to anomaly detection. These methods detect anomalies by checking outliers/deviations from the normal data distribution. However, the effectiveness of unsupervised anomaly detection algorithms is often unsatisfactory [13]. The false alarm rate of unsupervised models is higher, which requires much more effort for engineers to check the status of the cloud system.

- Lacking labels for supervised learning: As mentioned above, if the temporal property of time series data can be well incorporated into the labelled data, supervised machine learning methods such as SVM or Random Forest are good to be used to learn and predict anomaly patterns [29]. However, due to the scale and complexity of a cloud service system, labelling the whole dataset requires enormous human effort and is an almost impossible task. The problem of lacking labelled data limits the application of supervised anomaly detection methods to cloud service systems.

3 Proposed Approach

In order to address the challenges mentioned above, in this paper, we propose a novel time series anomaly detection method called ATAD (Active Transfer Anomaly Detection), which combines transfer learning and active learning technologies for anomaly detection in cloud monitoring data. Fig. 1 shows the overall workflow of ATAD.

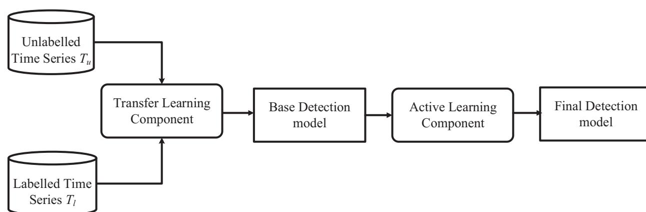


Figure 1: The overall workflow of ATAD

There are two sets of input data, one is the unlabelled time series data T_u on which anomaly detection will be conducted, the other is labelled time series data T_l collected from the public domains or the other components of the cloud system. In T_l, each point in the time series has been manually labelled as either anomaly or normal.

Our approach consists of two main components, namely *Transfer Learning component* and *Active Learning component*. In transfer learning, to incorporate temporal property of time series data, multiple general features are extracted from the raw dataset T_l and form the feature dataset F_l. Feature-based and instance-based transfer learning methods are then applied on F_l to learn a base detection model. After that, the unlabelled time series T_u goes through the same feature extraction process and forms the feature dataset F_u. The active learning component recommends a small number of informative samples from F_u for labelling through *Uncertainty and Context Diversity* (UCD) strategy. Then the labelled data is used to retrain the base detection model. After T rounds of active learning, we obtain the final anomaly detector.

We will describe more details about ATAD in the following sections.

3.1 Transfer Learning Component

Many machine learning methods assume that the distributions of labelled and unlabelled data are the same. However, transfer learning, in contrast, allows the domains, tasks, and distributions used in training and testing to be different [30]. In order to achieve knowledge transfer among different time series datasets, we utilize an efficient and effective transfer method for large-scale cloud monitoring data. For the anomaly detection problem in cloud systems, it is non-trivial to perform transfer learning directly. We need to consider the following factors when we design our transfer learning component.

- Cloud monitoring data is usually presented in the form of time series. Time series is not independent data, but a set of data points with temporal dependence. Thus the anomaly patterns of time series have contextual relevance. How to incorporate this relevance into anomaly detection is a challenge. In our work, we extract time series related features at each data point. Each data point is transformed from the original single-dimensional scalar into a high-dimensional feature vector, and the contextual information is preserved by these features. In ATAD, we not only take account of the simple descriptive features (statistical values), but also the order-aware features (forecasting error features and temporal features).

- For a time series, it is a problem what granularity transfer learning should be performed at. We can conduct transfer learning on entire time series, subseries, or discrete time points. If transfer learning is performed at a coarse-granularity (e.g. the entire time series or subseries) that contains several different anomalous patterns, it is not conducive to distinguish them. Further, coarse-granularity anomaly detection leads to the difficulty of locating and retrieving the cause of anomalies. In this work, we aim to conduct anomaly detection at a fine granularity (i.e. for each data point), and so we perform transfer learning at the level of data point.

- Transfer learning requires that the source domain and

the target domain have the underlying similarity. However, the time series generated from various components in a large-scale cloud system could be very different. We should guarantee that the source domain and the target domain come from similar services or have similar characteristics. Thus, during the transfer learning process, we need to filter out those source-domain samples that are not similar to the counterpart in the target domain.

Fig. 2 shows the workflow of the Transfer Learning Component. The following subsections describe our algorithm in detail.

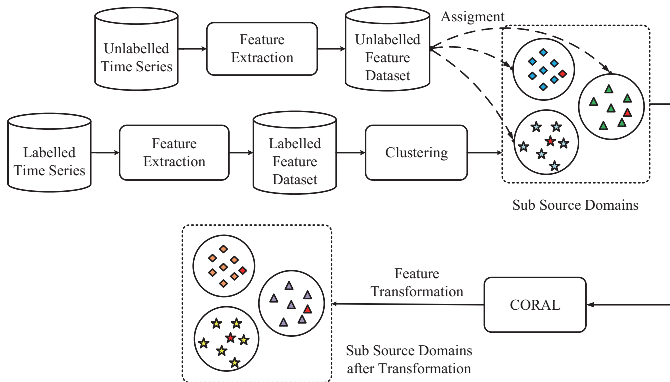


Figure 2: Transfer Learning Component

3.1.1 Feature Identification

The feature engineering process of ATAD converts each data point in a time series (T_l) into a set of features (F_l), which can capture both contextual and temporal information around the point. These features can be categorized into three groups: statistical features, forecasting error features, and temporal features.

Before computing the value of these features, we use Discrete Fourier Transform (DFT) to estimate the period p of the most dominant frequency. Different periods determine different sizes of the sliding window used in the following process.

Statistical features: Statistical features describe some basic characteristics around each data point in time series. We hold a view that the statistical features are able to describe the basic characteristics of different time series generated from various sources in cloud systems, and it is conducive to detecting anomalies that violate the basic characteristics. For example, in an active-running computation intensive service, if the average CPU utilization over a time window tends to be low, it may be an indicator that part of the computation process on it halts unexpectedly.

We identify features for depicting statistical characteristics of time series data (as listed in Table 1). These descriptive features are all calculated in a rolling window derived from the period p. They can represent short-period aspects of time series data such as mean, variance, autocorrelation, trend, remainder, and stationary test.

Table 1: Statistical Features

Feature	Description
Mean	Mean.
Var	Variance.
Crossingpoint[18]	The number of crossing points.
ACF1	First order of autocorrelation.
ACFremainder	Autocorrelation of remainder.
Trend	Strength of trend.
Linearity [18]	Strength of linearity computed on trend of STL [5] decomposition.
Curvature [18]	Strength of curvature computed on trend of STL [5] decomposition.
Entropy [18]	Spectral entropy [12].
ARCHtest.p [9]	P value of Lagrange Multiplier (LM) test for ARCH model [8].
GARCHtest.p [24]	P value of Lagrange Multiplier (LM) test for GARCH model [8].

Forecasting error features: Following the prior work [22], we use a set of error metrics resulted from time series forecasting as features. The intuition is that if the value of current point deviates from the forecasting result, there is more likely to be an anomaly. We use ensemble models to carry out forecasting and apply different models based on the seasonality of the time series. The models leverage classical forecasting techniques, namely SARIMA [27], Holt [19], Holt-Winters [19], and STL [5] for seasonal data, and SARIMA, Holt, Holt-Winters, and Polynomial Regression [15] for non-seasonal data, respectively. The metric *RMSE* (Root Mean Squared Error) is used to endow different forecasting methods with different weights at a fixed time. More weight should be assigned to more precise forecasting model. The weighted prediction result of the ensemble model from M models at time t is calculated by:

$$\hat{Y}_t = \sum_{m=1}^{M} \frac{\hat{Y}_{m,t}}{M-1} \cdot \left(1 - \frac{RMSE_{m,t}}{\sum_{n=1}^{M} RMSE_{n,t}}\right) \tag{1}$$

where $\hat{Y}_{m,t}$ is the prediction by model m at time t, $RMSE_{m,t}$ is the prediction error of model m at time t, $\hat{Y}_t$ is the ensemble prediction at time t.

After gaining the ensemble prediction $\hat{Y}_t$, we calculate 5 metrics on 3 rolling time windows to measure the bias between predicted and actual values. The metrics are shown

in Table 2, where $\hat{Y}$ is the predicted value, Y represents the actual value, and N is the size of the time window.

Table 2: Metrics used as forecasting error features

Features	Formula	Description
ME	$\frac{\sum(Y_i-\hat{Y}_i)}{N}$	Mean Error.
$RMSE$	$\sqrt{\frac{\sum(Y_i-\hat{Y}_i)^2}{N}}$	Root Mean Squared Error.
MAE	$\frac{\sum\lvert Y_i-\hat{Y}_i\rvert}{N}$	Mean Absolute Error.
MPE	$\frac{1}{N}\cdot\sum\frac{Y_i-\hat{Y}_i}{Y}$	Mean Percentage Error.
$MAPE$	$\frac{1}{N}\cdot\sum\frac{\lvert Y_i-\hat{Y}_i\rvert}{Y}$	Mean Average Percentage Error.

Temporal features: Generally speaking, the drastic changes of system metrics are likely to be anomalies. For example, the sharp decline of disk I/O traffic rate may be caused by the hardware failure in the disk array. To understand the changes of time series data over time, we identify temporal features (as shown in Table 3) by comparing data in two consecutive windows. We also compute the difference between current values and previous w values (e.g., the difference between x_{n-w} and x_n). In our implementation, we set $w = p/2, p, 2p, w_{pr}, w_{pr}/2$, respectively to get the corresponding different values (Diff-w). w_{pr} is selected according to some prior knowledge. For example, if a time series is recorded by hours, we can set $w_{pr} = 24$.

Table 3: Temporal Features

Features	Description
Max_level_shift	Max trimmed mean between two consecutive windows.
Max_var_shift	Max variance shift between two consecutive windows.
Max_KL_shift	Max shift in Kullback-Leibler divergence between two consecutive windows.
Lumpiness	Changing variance in remainder.
Flatspots	Discretize time series values into ten equal-sized intervals. Find maximum run length within the same bucket. [18].
Diff-w	The differences between the current value and the w-th previous value.

In summary, with original time series value, we extract total 37 features used to capture the characteristics of time series data. It is also worth mentioning that all those features are normalized in order to make them comparable among different time series.

3.1.2 The Transfer between Source Domain and Target Domain

In order to transfer knowledge between the source and target domains, it is necessary to narrow the difference between the two domains. Considering the effectiveness and efficiency requirements in the anomaly detection task, we propose a transfer method combining the instance-based transfer learning and feature-based transfer learning.

In transfer learning, we should guarantee that the source domain and the target domain come from similar fields (such as similar monitoring data) or own similar characteristics (such as trend or period). However, the source domain may consist of various time series data. Thus, the first step of transfer learning is to collect the time series data from the source domain that are similar to the data in the target domain. In ATAD, we use instance-based method to filter out those source-domain samples which are not similar to the counterpart in the target domain.

The idea of instance-based transfer learning is to select the source-domain samples which are similar to samples in the target domain so that the difference between two domains can be reduced. For source domain, after identifying features and converting T_l into F_l, we perform *K-means* [29] algorithm on F_l to build K clusters. Each cluster F_l^i $i \in [1,K]$ is a subset of F_l without overlap, which can be regarded as a sub source domain. To select the similar samples, firstly, the same feature extraction process is applied to the unlabelled time series data T_u to form feature dataset F_u. Then we calculate the Euclidean distance between each unlabelled sample and the central point of each cluster. Each target-domain sample will be assigned to the nearest sub source domain F_l^i. We denote the testing samples which are assigned into the same cluster as F_u^i $i \in [1,K]$.

After the instance-based transfer, we need to further narrow the difference between target domains and corresponding sub source domains from the perspective of feature space, because the distribution of features may still remain different. In ATAD, we conduct CORrelation ALignment (CORAL) [37] on each cluster, which is the idea of the feature-based transfer learning process. CORAL is a domain adaption algorithm, which can align the second-order statistics, namely, the co-variance of the source and target features in an unsupervised manner. Specifically, CORAL aims to minimize the Frobenius norm between co-variance matrices of the target and source domains, as shown in Eq. 2.

$$\min_A \left\| A^T C_l^i A - C_u^i \right\|_F^2 \tag{2}$$

where A is a linear transformation matrix, C_l^i is the co-variance matrix of labelled data in cluster i, and C_u^i is the co-variance matrix of unlabelled data in cluster i. We can get the optimal solution of A by whitening source data and re-coloring with target co-variance method, i.e. CORAL. More

details can be referred in [37]. Finally, we can get the new sub source domain features data $\hat{F}_l^i$ after transformation.

In the last step, we train a base supervised model, like Random Forest (RF) or Support Vector Machine (SVM), on each sub source domain $\hat{F}_l^i$. In the end, we can get K independent base models.

Some notes about the proposed transfer learning component:

- Base model: we use Random Forest as the supervised machine learning model (i.e. the base model) in our implementation. Random Forest can be implemented in a parallel way thus it owns high efficiency.
- Computational framework: we emphasize that this component can be regarded as a computational framework. In fact, the choices of distance measurement, clustering method, and base model are flexible.
- Assignment complexity analysis: when assigning unlabelled samples, we need to calculate the distance between each sample in the unlabelled set and the center points of all clusters (sub source domains). This time complexity is $\mathcal{O}(m \cdot K)$, where K is the number of clusters and m is the size of F_u. K is generally much less than m and can be regarded as a constant, so $\mathcal{O}(m \cdot K) \approx \mathcal{O}(m)$. Therefore, our assignment process has linear complexity.
- Parallel processing: it is worth noting that the sub source domains are completely independent to each other, which means that the follow-up processes for each sub source domain could be conducted in a parallel manner. This can help improve the efficiency of anomaly detection.

3.2 Active Learning Component

Due to the high complexity of the cloud service systems, the time series generated from different components are characterized by great diversity. Thus, transfer learning technique is not enough to achieve satisfactory results on various time series in cloud. In ATAD, leveraging active learning method, the diversified data with specific characteristics can be addressed with a small amount of labelling effort.

Active learning focuses on minimizing the labeling effort of users and improving the accuracy of the prediction model. In this work, we utilize an active learning method that considers *Uncertainty* and *Context Diversity* of samples during sampling. We call it UCD for short.

3.2.1 Uncertainty

Most active learning methods use *uncertainty* as the principle to select samples for labelling [33] because it is believed that if a model is less certain about the classification results of some samples, labelling such samples would be more helpful to the base model. In our approach, we use the base model (Random Forest) to estimate the probability of an unlabelled data to be normal or anomalous. We then use the following formula to calculate the uncertainty for unlabelled samples:

$$Uncertainty = -\left|Prob(Normal) - Prob(Anomaly)\right| \quad (3)$$

where *Prob* represents the probability given by the base model.

We calculate the uncertainty according to Eq. 3 and sort them in descending order. The larger the uncertainty, the more it needs to be labelled.

3.2.2 Context Diversity

To recommend samples to be labelled, *diversity* is also an important factor to be considered. Sometimes, two samples are very similar or may belong to the same anomaly pattern. It is unnecessary to label both of them.

Traditional diversity methods are generally based on clustering [7], which do not consider the context of samples in time series scenario. In cloud systems, time series of system metrics, such as CPU utilization or traffic load, are continuous without drastic breakpoints. Thus, samples that are adjacent in time series tend to be similar. Our active learning algorithm makes full use of this property in time series.

Specifically, we sort all samples by uncertainty and scan them sequentially. If a new sample we scanned is in the context of another sample in the candidate set, i.e. these two samples are adjacent to each other, we hold a view that the information embedded in them could also be similar. We thus ignore the new sample because it may contain redundant information. If the new sample does not appear in the context of all samples in the candidate set, it is added to the candidate set. In our work, the context of sample x_t is controlled by a parameter α, which represents a range from $x_{t-\alpha}$ to $x_{t+\alpha}$ in a time series. Fig. 3 illustrates the concept of context diversity in a time series. More details can be found in Algorithm 1.

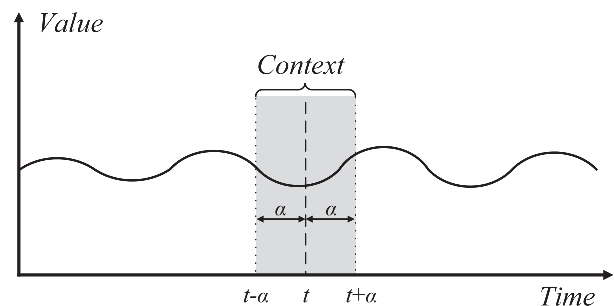


Figure 3: Context in time series

For each source domain, we perform active learning on its own testing data F_u^i. We recommend d diverse and uncertain samples to be labelled, and add the labelled samples into the training set to retrain the base model. After repeating T rounds of this process, we obtain the final detection model.

Algorithm 1: Active Learning Component

Input:
labelled feature data from source domain F_l;
unlabelled feature data from target domain F_u;
base model M_{base} obtained from F_l;
the number of samples at each round d;
context parameter α;
the number of rounds T;

Output: Final model M_{final};

```
1  M = M_base
2  for i = 1 to T do
3      Candidate Set, S = ∅
4      Prob(Normal), Prob(Anomaly) = M(F_u)
5      Uncertainty =
         −|Prob(Normal) − Prob(Anomaly)|
6      Uncertainty_Candidate = argsort(Uncertainty)
7      for j = 1 to S.size() do
8          if S == ∅ then
9              S = S ∪ Uncertainty_Candidate[j]
10             continue
11         end
12         if S.size() > d then
13             break
14         end
15         if Uncertainty_Candidate[j] ∉
             [x_{t−α}, x_{t+α}], ∀x_t ∈ S then
16             S = S ∪ Uncertainty_Candidate[j]
17         end
18     end
19     label the S as F_fed
20     F_l = F_l ∪ F_fed
21     M = M.train(F_l)
22 end
23 return M as M_final
```

3.3 Usage of ATAD

Transfer learning and active learning are only conducted in the training process. Once the training of ATAD finished, a classifier will be generated and further applied to anomaly detection task in practice. The detection process of ATAD is as follows: Firstly, we feed the time series data to be detected into the feature extraction component and extract features as the training process does. After that, the features are input to the trained classifier to get the anomaly probabilities. Finally, the points whose probabilities are higher than a pre-specified threshold are predicted as anomalies. This pre-specified threshold can be treated as the sensitivity parameter for adapting to different requirements of various users and scenarios.

4 Experiments

In this section, we evaluate the effectiveness of our approach ATAD through a series of experiments. We aim to answer the following research questions in evaluation:

RQ1: How effective is the proposed ATAD approach?

RQ2: How effective is the Transfer Learning component?

RQ3: How effective is the Active Learning component?

RQ4: How effective is ATAD in detecting anomalies in a company's local dataset based on public datasets?

4.1 Dataset and Setup

We use two public time series anomaly detection datasets, NAB [23] and Yahoo [22], to evaluate our proposed method. NAB is a novel benchmark for evaluating anomaly detection algorithms in streaming, real-time applications. It contains datasets collected from different fields, including AWS, Twitter, and Artificial, etc. Each dataset contains several time series of variable length. The AWS dataset contains different server metrics, such as CPU utilization, network traffic, disk write bytes, etc, collected by the Amazon CloudWatch service. The Artificial dataset contains artificially generated time series data with various types of anomalies, while the anomaly patterns are much simpler. The Twitter dataset is the collection of Twitter mentions of large publicly-traded companies such as Google and IBM. The Yahoo dataset consists of metrics of various Yahoo services, which reflects the status of Yahoo system. All datasets are given in time series form and every data point is manually labelled. These time series range in length from hundreds to thousands. The proportion of anomaly is about 1% $\sim$ 5%.

In the experiments, we use Yahoo, AWS, Artificial and Twitter datasets as the testing set (target domain). There are two reasons for this choice. First, these datasets are related to cloud monitoring data. Second, the scale of these datasets are relatively large, or the anomalous points are also much more than other datasets. More details about datasets are shown in Table 4. The first column is the average length of time series. The second and third columns are the total number of data points and the number of anomalies, respectively. We also show the percentage of anomaly data points in the last column.

We perform cross-dataset anomaly detection according to our setup. The experiments are conducted on four pairs of datasets, including non-Yahoo→Yahoo, non-AWS→AWS, non-Twitter→Twitter, and non-Artificial→Artificial. The right side of the arrow represents the unlabelled testing

Table 4: Summary of datasets

Dataset	time series mean length	#data points	#anomaly points	%anomaly
Yahoo	1415	92016	1617	1.76%
AWS	3985	67740	3097	4.57%
Artificial	4032	16128	624	3.87%
Twitter	15862	142765	217	0.15%

dataset, i.e. target domain and the left side of the arrow represents the labelled dataset from other fields. The labels of the target domain are not used during the training and transfer learning process. They are only used during active learning and evaluation.

4.2 Evaluation Metric

We evaluate the accuracy of anomaly detection methods using F1-Score, which is defined as follows:

$$F1 = \frac{2 \cdot P \cdot R}{P + R}, \quad P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN} \tag{4}$$

where P and R denote the precision and recall, respectively. In addition, TP, FP, FN, and TN are referred to as true positive, false positive, false negative, and true negative, respectively. We might fail to detect potential anomalies if only focus on the precision. On the other hand, a couple of false positives might be received when we solely pay attention to the recall. F1-Score builds up the balance of the precision and recall, is therefore used as the main evaluation metric in our experiments.

Under an acceptable recall, we expect the anomaly detector to achieve as high precision as possible. The more precise the detector is, the less amount of false alarms will be reported, and thus less human effort is required to investigate. Therefore, precision can be regarded as an important metric, which reflects the automation degree of anomaly detection systems.

4.3 Results

4.3.1 RQ1: How effective is ATAD?

In this section, we evaluate the effectiveness of our proposed approach, ATAD. First, we compare ATAD with some commonly-used anomaly detection algorithms to examine the superior performance of the proposed approach. Second, we present the advantages of ATAD in saving labelling cost, as a comparison with supervised learning based anomaly detectors.

The comparative anomaly detection algorithms are developed based on Isolation Forest (iForest) [26], K-Sigma [14], Seasonal Hybrid ESD (S-H-ESD) [16] and Random Forest (RF). The iForest model ensembles random split tree models to identify which points are isolated. K-Sigma is a common statistics-based method, in which the samples are taken as anomalies whose values deviate more than k times of the variance of samples from the corresponding mean. S-H-ESD builds upon the Generalized ESD test [32], and is able to detect both global and local anomalies. This algorithm is incorporated in the well-known *AnomalyDetection* R package [39] and thus is widely used. Because the RF is used in ATAD as the based learning classifier, we exploit a classical RF based supervised model as a comparison to demonstrate the superior performance of ATAD. The RF and iForest models use the same features as the ATAD. The K-Sigma and S-H-ESD are performed on the raw time series.

In the experiments, we evaluate the metrics under different settings and present the best results in terms of F1-Score in the following. Towards this end, the proportion of anomalies in iForest is set to 0.01, 0.05, 0.10 and 0.20, respectively, and the integer k varies from 1 to 3 in K-Sigma. In addition, the maximum proportion of anomalies in S-H-ESD is set to 0.01 and 0.05, respectively. We labelled the same proportion of samples for RF as the counterpart in ATAD used in the target domain. In ATAD, the K in the Transfer Learning component changes from 3 to 5, whereas the number of rounds T and the labelling ratio are fixed to 3 and 1%, respectively, in the Active Learning component. The probability threshold is set to $0.6 \sim 0.8$.

The results are shown in Table 5. It is clear that the ATAD can achieve much higher F1-Scores than other approaches on all datasets. Particularly, though the supervised learning method (RF) achieves better performance than those unsupervised methods, the ATAD outperforms the RF when given the same number of labels.

In order to demonstrate the advantages of ATAD in saving labelling efforts, we compare the number of labelled samples of ATAD and RF under the similar F1-Scores. The relevant results are presented in Table 6, whose first and third column depicted the F1-Scores of the supervised model (RF) and the ATAD, respectively. Their corresponding quantities of labelled data are included in the second and fourth column of Table 6. It is evident that the supervised model generally takes $3 \sim 10$ times more labels than the ATAD to achieve comparable results. The superior performance benefits the transferred knowledge from the source domain in the Transfer Learning component. As a consequence, a small number of labels is required in the target domain. Moreover, the UCD method helps to further reduce the number of labels because the performance can be improved rapidly and significantly by the extraordinary informative recommendations in the Active Learning component.

Table 5: Results of Comparative Methods

Dataset	Method	Precision	Recall	F1-Score
Non-Yahoo → Yahoo	iForest	0.3832	0.2183	0.2781
	K-Sigma	0.6499	0.3364	0.4433
	S-H-ESD	0.2779	0.6215	0.3840
	RF	0.8668	0.2075	0.3348
	ATAD	0.8847	0.4040	**0.5547**
Non-AWS → AWS	iForest	0.1523	0.0491	0.0743
	K-Sigma	0.6899	0.1992	0.3091
	S-H-ESD	0.5382	0.7100	0.6123
	RF	0.9999	0.6226	0.7674
	ATAD	0.9195	0.8142	**0.8637**
Non-Artificial → Artificial	iForest	0.3477	0.9006	0.5017
	K-Sigma	1.000	0.1730	0.2950
	S-H-ESD	0.7888	0.4568	0.5785
	RF	0.9182	0.9301	0.9241
	ATAD	0.9990	0.9850	**0.9924**
Non-Twitter → Twitter	iForest	0.4685	0.3087	0.3722
	K-Sigma	0.2608	1.0000	0.2608
	S-H-ESD	0.7481	0.4654	0.5739
	RF	0.7285	0.4811	0.5795
	ATAD	0.8769	0.6951	**0.7755**

Table 6: Supervised Model (Random Forest) vs. ATAD

	RF	#labels	ATAD	#labels
Non-Yahoo → Yahoo	0.5440	4600	0.5547	920
Non-AWS → AWS	0.8403	763	0.8637	254
Non-Artificial → Artificial	0.9755	1612	0.9924	161
Non-Twitter → Twitter	0.7126	17131	0.7755	1427

4.3.2 RQ2: How effective is the Transfer Learning Component?

We evaluate the effectiveness of our Transfer Learning Component from the following two aspects:

- The effectiveness of identified features, including forecasting error, statistical, and temporal features.
- The effectiveness of our proposed transfer method.

Effectiveness of the identified features Our proposed transfer learning is based on many time series features including forecasting error, statistical, and temporal features. The conventional transfer learning is only based on statistical features such as averages and variances [30]. The statistical features are simple descriptive values that are independent of the context of time series.

In order to evaluate the validity of features used in ATAD, we perform an experiment on four dataset pairs. We compare ATAD using statistical features alone and ATAD using order-aware features (forecasting error features and temporal features).

Experimental results are shown in Table 7. It can be seen that using statistical features alone for ATAD leads to poor results, and when order-aware features are added, the performance becomes better. The reason is that the transfer learning needs to narrow the differences between the source domain and the target domain. If only statistical features are extracted and the characteristics of time series are ignored, the features do not reflect the context property in time series. Thus the resulting performance is less satisfactory.

Table 7: The effectiveness of features (F1-Score)

Features	Yahoo	AWS	Artifical	Twitter
Statistical	0.2956	0.7387	0.7441	0.6937
Order-aware	0.4200	0.8441	0.7569	0.6622
All features	**0.5781**	**0.8637**	**0.9924**	**0.7755**

Random Forest also provides a popular approach to feature ranking. Here we use the Random Forest classifier to evaluate the importance of the features used in ATAD. The importance of a feature can be ranked according to the mean decrease impurity [3]. In Table 8, we show the top-10 most important features of the RF models in ATAD. The definitions of these features can be found in Table 1, Table 2 and Table 3. We can see that the forecasting error features and the original time series are much more informative for anomaly detection. In addition, some temporal features, such as Diff-*w* and Flatspots, also play an important role in the model. It is also worth noting that the important features of each dataset are different. It implies that we should consider all the features comprehensively when conducting transfer learning between different datasets.

Effectiveness of the proposed transfer method In our transfer learning, we create multiple independent sub source domains through clustering and conduct the CORAL algorithm to transform the features of sub source domains. We expect the transfer learning can reduce the labelling effort for users, because we can activate the Active Learning component directly based on the transfer model, without any manual labels in the testing set. We validate the effectiveness of our Transfer Learning component by comparing with

Table 8: Feature Importance Evaluation

Dataset	Important Features
Yahoo	Original_data, RMSE, MAE, ME, Mean, MPE, Diff-p, Diff-$w_{pr/2}$, Diff-2p, MAPE
AWS	Original_data, RMSE, MAE, ME, Mean, ACF1, Diff-2p, Curvature, Flatspots, Diff-p
Artificial	Original_data, Mean, Diff-p, MPE, Flatspots, RMSE, MAE, Diff-2p, Diff-w_{pr}, Lumpiness.
Twitter	Var, RMSE, MAE, ARCHtest.p, Mean, Flatspots, Original_data, Max_level_shift, MPE, Original Data

naive active learning applied directly on target domain without transfer methods.

Specifically, in the naive method, we pre-fetch a part of samples in the testing dataset to train a base model. After that, we use this base model to conduct the active learning process. This naive active learning approach needs no auxiliary public labelled data as source domain and transfer learning technique, but it needs more labelling effort for building the base model, as illustrated in Table. 9.

Table 9: Comparative Experiment of ATAD and Naive Active Learning without Transfer Learning (F1-Score)

	Naive		ATAD	
	F1-Score	#labels	F1-Score	#labels
Yahoo	0.5691	1380	0.5697	920
AWS	0.8589	381	0.8637	254
Artificial	0.9815	322	0.9924	161
Twitter	0.6164	2855	0.7755	1427

In Table 9, we fix the labelling ratio of the active learning process to 1% in both Naive method and ATAD. For fairness, in the naive method, we pre-fetch samples from the testing set in stratified style to train the base model. From the table, it can be seen that the number of samples required for the naive method without transfer learning component is, in average, 1.56 times than that of ATAD, but its results are still slightly lower than those achieved by ATAD. We also illustrate the number of labels required by the supervised model, naive active learning model without transfer methods, and ATAD in Fig 4. It is clear that active learning can significantly save labelling effort and ATAD can further reduce the labelling cost by introducing the transfer learning component.

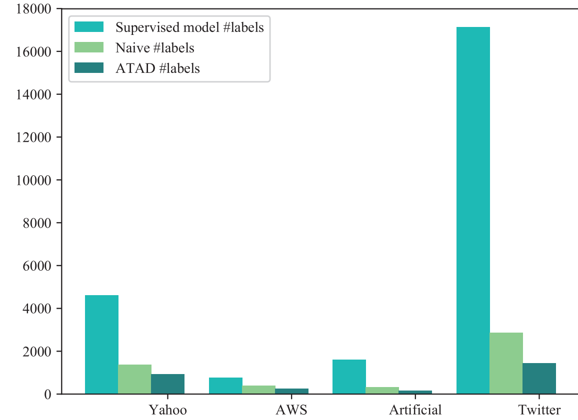


Figure 4: The number of labels required by Supervised Model, Naive Active Learning without transfer learning and ATAD

Table 10: The experimental result with active learning alone (F1-Score)

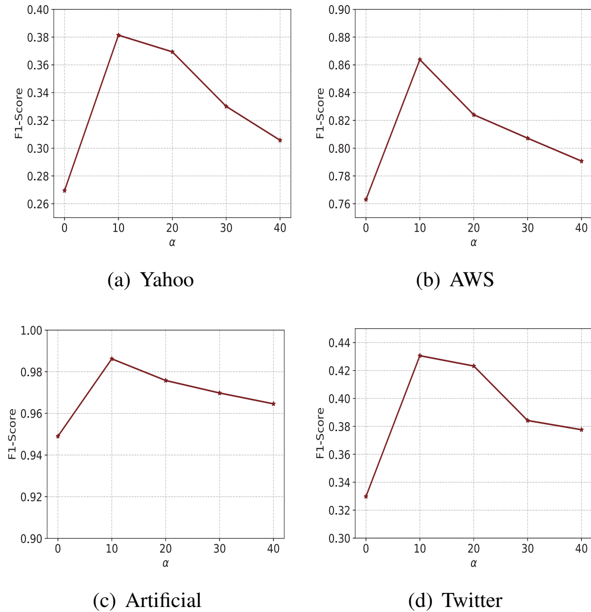
Dataset	Method	Base	Round1	Round2	Round3
Yahoo	U	0.2090	0.2437	0.2839	0.2695
	UCD	0.2090	0.2383	0.3032	**0.3814**
	random	0.2090	0.2275	0.2256	0.2196
AWS	U	0.0443	0.1932	0.6838	0.6799
	UCD	0.0443	0.7550	0.8092	**0.8879**
	random	0.0443	0.3227	0.5164	0.6604
Artificial	U	0.6239	0.7272	0.8737	0.9006
	UCD	0.6239	0.7340	0.8780	**0.9715**
	random	0.6239	0.6446	0.6275	0.5000
Twitter	U	0.1647	0.2916	0.2959	0.3298
	UCD	0.1999	0.3070	0.3703	**0.4232**
	random	0.1647	0.1798	0.1944	0.1689

4.3.3 RQ3: How effective is the Active Learning component?

To evaluate the effectiveness of the Active Learning component, we use total labelled datasets F_l to train the base model and do not apply transfer learning on them. We compare the UCD method with the conventional Uncertainty method (U) and the random selecting method (random). In all experiments, we conduct 3 rounds of active learning and select 60 samples for labelling at each round. In order to avoid the data leakage problem [20], all samples labelled by the active learning component are removed from the testing set. The experimental results are shown in Table 10.

From Table 10, it can be seen that the UCD method achieves the best results on all datasets, confirming the usefulness of incorporating time series context diversity. We

also find that as the number of rounds increases, the F1-Scores achieved by UCD are also steadily improved. However, the random selection method cannot guarantee such trend. The results confirm the validity of the active learning method.



(a) Yahoo (b) AWS

(c) Artificial (d) Twitter

Figure 5: Experimental results with different α

The hyper-parameters in the Active Learning component include the number of samples to be labelled per round d, the number of rounds T, and the context parameter α. Obviously, if T and d are much larger, the entire algorithm will be closer to supervised learning. Therefore the accuracy will be improved gradually.

We conduct an experiment to explore the impact of different α values on the final results. In Fig. 5, we can see that as the α value increases, the F1-Score rises rapidly first and then falls gradually. The larger α means the wider range of context and more attention to Context Diversity because the wider context will cause more uncertain candidate samples to be discarded. The smaller α means the narrower range of context and more attention to Uncertainty. If $\alpha = 1$, UCD becomes purely U. Therefore, α can be treated as a parameter which trades off between Context Diversity and Uncertainty. Too large or too small α value will result in loss of accuracy. In conclusion, choosing an appropriate α can make full use of both kinds of information. Through our practice on all experimental datasets, we empirically set $\alpha = 10$.

4.3.4 RQ4: How effective is ATAD in detecting anomalies in a company's local dataset based on public datasets?

In RQ1, we examined the performance of ATAD on public datasets. In this RQ, we evaluate its effectiveness by using practical industrial data from the large-scale cloud system in Microsoft. More specifically, we hourly record IOPS (I/O Operations Per Seconds) of the storage service in a cluster and collect the corresponding time series data in the past few months. The data is labelled by domain experts in the operation team. All public datasets are utilized as the source domain, and then the ATAD is applied to the target domain dataset, i.e., the IOPS data, for anomaly detection. The experiment is conducted as the similar process in RQ1.

Table 11: Experimental result on IOPS dataset of Microsoft

	Precision	Recall	F1-Score
iForest	0.2886	0.3988	0.3349
K-Sigma	0.8170	0.1882	0.3059
S-H-ESD	0.9117	0.1741	0.2924
RF	0.5213	0.6724	0.5873
ATAD	0.8082	0.6188	**0.7009**

The results are shown in Table 11. It is noted that the traditional unsupervised methods cannot work well due to the low recalls and the resulting F1-Scores. The extremely low recalls exhibit a high probability in failing to detect potential anomalies, which might cause a huge customer and financial loss. RF can achieve a higher recall but lower precision, which means lots of false alarms. In contrast, ATAD can achieve a higher F1-Score, which is beneficial to improving the service availability and detection automation.

5 Threats to Validity

- Data quality: in this work, we use public datasets in evaluation. The labels about anomalies are provided with the datasets. Although these dataset are of high quality and are used by several other studies [23, 22], it is possible that they contain a small degree of noise. Furthermore, their data volume is also limited. We will experiment with larger scale datasets in our future work.

- Correctness of labelling: the Active Learning component in ATAD requires users to manually label a few percentages of data and assumes that the labels are correct. However, in reality, the quality of labeling may vary (i.e., users may incorrectly label a data point).

- Data leakage: in order to avoid the data leakage problem, we remove the samples labelled in the active learning process from the testing set. Since active learning

may recommend different samples to be labelled at different rounds, the resulting testing set could be slightly different in each experiment, which may cause bias in comparisons. However, the proportion of samples to be labelled is very small, so we can ignore this influence on the final results.

6 Related Work

In recent years, there have been many studies on anomaly detection problem for time series data. The proposed methods can be largely divided into supervised methods, semi-supervised methods, unsupervised methods and statistical-based methods [4].

Unsupervised methods do not require manual labelling. These methods assume that normal instances are far more frequent than anomalies and anomalies deviate from the normal data distribution. For example, Ahmad, et al. proposed an unsupervised online sequence memory algorithm called Hierarchical Temporal Memory to detect anomaly in streaming data [2]. Xu, et al. proposed Donut, an unsupervised anomaly detection algorithm based on Variational Auto Encoder (VAE) [41].

Supervised methods aim to build a classification model for normal and anomaly classes. General supervised machine learning models can be applied to this problem. Steinwart, et al. interpreted anomaly detection problem as a binary classification task and proposed a supervised framework [36]. The most famous is the anomaly detection system of Yahoo, called EGADS [22], which used a collection of anomaly detection and forecasting models with an anomaly filtering layer for accurate and scalable anomaly detection on time series. Opperentice [25] used operators' periodical labels on anomalies to train a random forest classifier and automatically select parameters and thresholds.

Semi-supervised methods assume that the training data has labelled instances for only the normal class. Malhotra, et al. used stacked LSTM networks trained on non-anomalous data as a predictor to detect anomaly [28]. Erfani, et al. used a combination of one-class SVM model and deep learning model to detect anomaly [10]. Daneshpazhouh, et al. presented an entropy-based method that consists of two phases, including reliable negative examples extraction and entropy-based outlier detection [6].

Statistical-based methods are built up based on the statistical theory. The most famous method is K-Sigma method [14], in which the samples are taken as anomalies whose values deviate more than k times of the variance of samples from the corresponding average. Recently, more advanced methods using Extreme Value Theory [34, 42] were proposed. Compared with K-Sigma method, these methods do not require prior assumption on the data distribution.

From the view of effectiveness, supervised and semi-supervised methods generally perform better than unsupervised methods and statistical-based methods. However, due to the high labelling cost, they are difficult to be applied in the real world when the scale of dataset is very large. More recently, transfer learning and active learning techniques are applied to deal with this important problem. Transfer learning has been widely applied in many fields like classification, regression, and forecasting [30]. For example, Spiegel, et al. embeds a given set of labelled data into dissimilarity space, leading to enriched feature representations that facilitate statistical learning procedures [35]. Vercruyssen, et al. transferred labelled examples from a source domain to a target domain where no labels are available and constructed a nearest-neighbor classifier in the target domain with DTW measure [40]. Another technique to reduce the labelling effort is active learning. For example, Abe, et al. used a selective sampling mechanism based on active learning to the reduced classification problem of outlier detection [1]. Pelleg, et al. proposed a novel active learning method to identify rare category records in an unlabelled noisy set with a small budget of data points that they are prepared to categorize [31].

In our work, we combine transfer learning and active learning methods so that we could achieve a balance between labelling effort and performance. On the one hand, unlike unsupervised models, ATAD could achieve a high F1-Score. On the other hand, we only need to label a few number of samples from the unlabelled dataset. These advantages are not possessed by the existing methods mentioned above.

7 Conclusion

In this paper, we propose a novel anomaly detection method ATAD for cloud service systems. ATAD combines transfer learning and active learning techniques. In transfer learning, we use an existing labelled dataset as the source dataset. We extract multiple features, construct source domains, and use the labelled data in each source domain to train base models for a target, unlabelled dataset. In active learning, we use the UCD method to recommend informative samples in the target dataset to label and retrain the base models. Our experiments on cross-dataset anomaly detection show that we can achieve satisfactory detection accuracy by labeling only a small number of samples in the target dataset. We have also evaluated the effectiveness of ATAD using real-world data collected from a production cloud system in Microsoft.

8 Acknowledgement

We thank Professor Mickey Gabel (University of Toronto) for the valuable and constructive suggestions on this paper.

References

[1] ABE, N., ZADROZNY, B., AND LANGFORD, J. Outlier detection by active learning. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 504–509.

[2] AHMAD, S., LAVIN, A., PURDY, S., AND AGHA, Z. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing 262* (2017), 134–147.

[3] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[4] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM computing surveys (CSUR) 41*, 3 (2009), 15.

[5] CLEVELAND, R. B., CLEVELAND, W. S., AND TERPENNING, I. Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics 6*, 1 (1990), 3.

[6] DANESHPAZHOUH, A., AND SAMI, A. Entropy-based outlier detection using semi-supervised approach with few positive examples. *Pattern Recognition Letters 49* (2014), 77–84.

[7] DASGUPTA, S., AND HSU, D. Hierarchical sampling for active learning. In *Proceedings of the 25th international conference on Machine learning* (2008), ACM, pp. 208–215.

[8] ENGLE, R. Garch 101: The use of arch/garch models in applied econometrics. *Journal of economic perspectives 15*, 4 (2001), 157–168.

[9] ENGLE, R. F. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica 50*, 4 (1982), 987–1007.

[10] ERFANI, S. M., RAJASEGARAR, S., KARUNASEKERA, S., AND LECKIE, C. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition 58* (2016), 121–134.

[11] GABEL, M., SCHUSTER, A., BACHRACH, R.-G., AND BJØRNER, N. Latent fault detection in large scale services. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* (2012), IEEE, pp. 1–12.

[12] G.GOREG. Forecastable component analysis. 64–72.

[13] GÖRNITZ, N., KLOFT, M., RIECK, K., AND BREFELD, U. Toward supervised anomaly detection. *Journal of Artificial Intelligence Research 46* (2013), 235–262.

[14] GRAFAREND, E. W. Linear and nonlinear models: Fixed effects, random effects, and mixed models. *Walter De Gruyter* (2006).

[15] HEIBERGER, R. M., AND NEUWIRTH, E. *Polynomial Regression*. Springer New York, New York, NY, 2009, pp. 269–284.

[16] HOCHENBAUM, J., VALLIS, O. S., AND KEJARIWAL, A. Automatic anomaly detection in the cloud via statistical learning. *arXiv preprint arXiv:1704.07706* (2017).

[17] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[18] HYNDMAN, R. J., WANG, E., AND LAPTEV, N. Large-scale unusual time series detection. In *Data Mining Workshop (ICDMW), 2015 IEEE International Conference on* (2015), IEEE, pp. 1616–1619.

[19] KALEKAR, P. S. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi School of Information Technology 4329008* (2004), 1–13.

[20] KAUFMAN, S., ROSSET, S., PERLICH, C., AND STITELMAN, O. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD) 6*, 4 (2012), 15.

[21] KAVULYA, S. P., DANIELS, S., JOSHI, K., HILTUNEN, M., GANDHI, R., AND NARASIMHAN, P. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* (2012), IEEE, pp. 1–12.

[22] LAPTEV, N., AMIZADEH, S., AND FLINT, I. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), ACM, pp. 1939–1947.

[23] LAVIN, A., AND AHMAD, S. Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on* (2015), IEEE, pp. 38–44.

[24] LEE, J. H. H. A lagrange multiplier test for garch models. *Economics Letters 37*, 3 (1991), 265–271.

[25] LIU, D., ZHAO, Y., XU, H., SUN, Y., PEI, D., LUO, J., JING, X., AND FENG, M. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the 2015 Internet Measurement Conference* (2015), ACM, pp. 211–224.

[26] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on* (2008), IEEE, pp. 413–422.

[27] MAKRIDAKIS, S., AND HIBON, M. Arma models and the box–jenkins methodology. *Journal of Forecasting 16*, 3 (1997), 147–163.

[28] MALHOTRA, P., VIG, L., SHROFF, G., AND AGARWAL, P. Long short term memory networks for anomaly detection in time series. In *Proceedings* (2015), Presses universitaires de Louvain, p. 89.

[29] NASRABADI, N. M. Pattern recognition and machine learning. *Journal of electronic imaging 16*, 4 (2007), 049901.

[30] PAN, S. J., AND YANG, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering 22*, 10 (2010), 1345–1359.

[31] PELLEG, D., AND MOORE, A. W. Active learning for anomaly and rare-category detection. In *Advances in neural information processing systems* (2005), pp. 1073–1080.

[32] ROSNER, B. Percentage points for a generalized esd many-outlier procedure. *Technometrics 25*, 2 (1983), 165–172.

[33] SETTLES, B. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning 6*, 1 (2012), 1–114.

[34] SIFFER, A., FOUQUE, P.-A., TERMIER, A., AND LARGOUET, C. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), ACM, pp. 1067–1075.

[35] SPIEGEL, S. Transfer learning for time series classification in dissimilarity spaces. *Proceedings of AALTD* (2016), 78.

[36] STEINWART, I., HUSH, D., AND SCOVEL, C. A classification framework for anomaly detection. *Journal of Machine Learning Research 6*, Feb (2005), 211–232.

[37] SUN, B., FENG, J., AND SAENKO, K. Return of frustratingly easy domain adaptation. In *AAAI* (2016), vol. 6, p. 8.

[38] TERZI, D. S., TERZI, R., AND SAGIROGLU, S. Big data analytics for network anomaly detection from netflow data. In *Computer Science and Engineering (UBMK), 2017 International Conference on* (2017), IEEE, pp. 592–597.

[39] TWITTER. Anomalydetection. `https://github.com/twitter/AnomalyDetection`, 2015.

[40] VERCRUYSSEN, V., MEERT, W., AND DAVIS, J. Transfer learning for time series anomaly detection. *IAL@ ECML PKDD 2017*, 27.

[41] XU, H., CHEN, W., ZHAO, N., LI, Z., BU, J., LI, Z., LIU, Y., ZHAO, Y., PEI, D., FENG, Y., ET AL. Unsupervised anomaly detection via variational autoencoder for seasonal kpis in web applications. *arXiv preprint arXiv:1802.03903* (2018).

[42] YUANYAN, L., XUEHUI, D., AND YI, S. Data streams anomaly detection algorithm based on self-set threshold. In *Proceedings of the 4th International Conference on Communication and Information Processing* (2018), ACM, pp. 18–26.

[43] ZIMEK, A., AND SCHUBERT, E. Outlier detection. *Encyclopedia of Database Systems* (2017), 1–5.